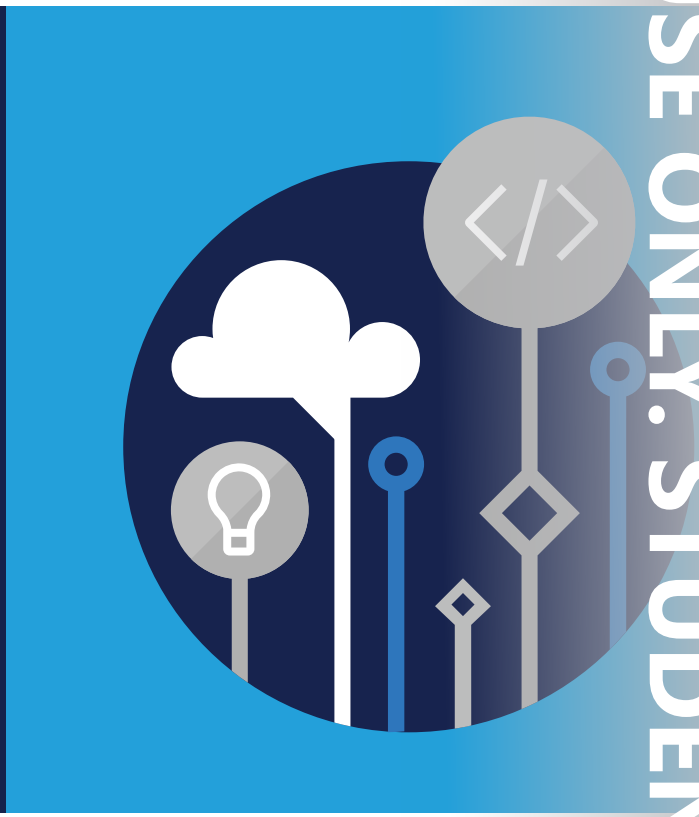
Microsoft

Microsoft Official Course



AZ-203T05

Monitor, troubleshoot, and optimize Azure solutions

AZ-203T05

Monitor, troubleshoot, and optimize Azure solutions

Contents

Module 0 Welcome to the course

Start Here

Welcome

Welcome to the **Monitor, troubleshoot, and optimize Azure solutions** course. This course is part of a series of courses to help you prepare for the **AZ-203: Developing Solutions for Microsoft Azure**[1] certification exam.

Candidates for this exam are Azure Developers who design and build cloud solutions such as applications and services. They participate in all phases of development, from solution design, to development and deployment, to testing and maintenance. They partner with cloud solution architects, cloud DBAs, cloud administrators, and clients to implement the solution.

Candidates should be proficient in developing apps and services by using Azure tools and technologies, including storage, security, compute, and communications.

Candidates must have at least one year of experience developing scalable solutions through all phases of software development and be skilled in at least one cloud-supported programming language.

Exam study areas

AZ-203 includes six study areas, as shown in the table. The percentages indicate the relative weight of each area on the exam. The higher the percentage, the more questions you are likely to see in that area.

AZ-203 Study Areas	Weight
Develop Azure Infrastructure as a Service compute solutions	10-15%
Develop Azure Platform as a Service compute solutions	20-25%
Develop for Azure storage	15-20%
Implement Azure security	10-15%

1 https://www.microsoft.com/en-us/learning/exam-az-203.aspx

AZ-203 Study Areas	Weight
Monitor, troubleshoot, and optimize Azure solutions	15-20%
Connect to and consume Azure, and third-party, services	20-25%

✓ This course will focus on preparing you for the **Monitor, troubleshoot, and optimize Azure solutions** area of the AZ-203 certification exam.

Course description

In this course students will gain the knowledge and skills needed to ensure applications hosted in Azure are operating efficiently and as intended. Students will learn how Azure Monitor operates and how to use tools like Log Analytics and Application Insights to better understand what is happening in their application. Students will also learn how to implement autoscale, instrument their solutions to support monitoring and logging, and use Azure Cache and CDN options to enhance the end-user experience.

Throughout the course students learn how to create and integrate these resources by using the Azure Portal, Azure CLI, REST, and application code.

Level: Intermediate

Audience:

- Students in this course are interested in Azure development or in passing the Microsoft Azure Developer Associate certification exam.
- Students should have 1-2 years experience as a developer. This course assumes students know how to code and have a fundamental knowledge of Azure.
- It is recommended that students have some experience with PowerShell or Azure CLI, working in the Azure portal, and with at least one Azure-supported programming language. Most of the examples in this course are presented in C# .NET.

Course Syllabus

Module 1: Introduction to Azure Monitor

- Overview of Azure Monitor

Module 2: Develop code to support scalability of apps and services

- Implement autoscale
- Implement code that addresses singleton application instances
- Implement code that handles transient faults

Module 3: Instrument solutions to support monitoring and logging

- Configure instrumentation in an app or server by using Application Insights
- Analyze and troubleshoot solutions by using Azure Monitor

Module 4: Integrate caching and content delivery within solutions

- Azure Cache for Redis
- Develop for storage on CDNs

Module 1 Introduction to Azure Monitor

Overview of Azure Monitor

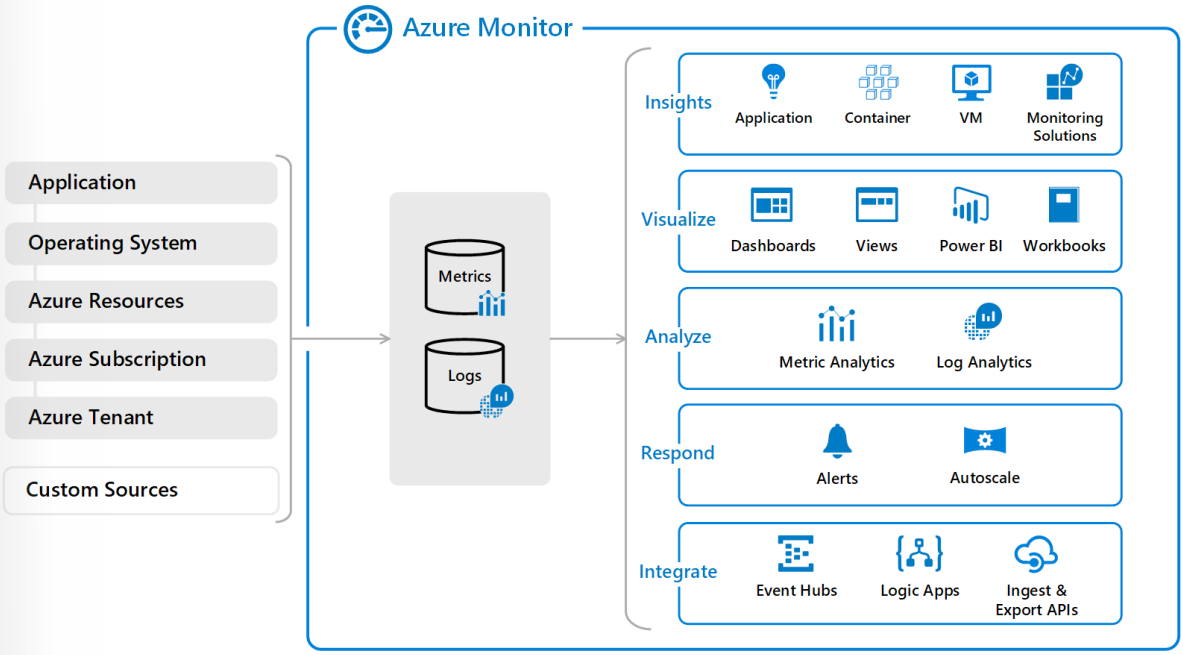
Azure Monitor overview

Note: Consolidation of monitoring services into Azure Monitor

Log Analytics and Application Insights have been consolidated into Azure Monitor to provide a single integrated experience for monitoring Azure resources and hybrid environments.

Overview

The following diagram gives a high-level view of Azure Monitor. At the center of the diagram are the data stores for metrics and logs which are the two fundamental types of data use by Azure Monitor. On the left are the sources that collect telemetry from different monitored resources and populate the data stores. On the right are the different functions that Azure Monitor performs with this collected data such as analysis, alerting, and streaming to external systems.



Monitoring data platform

All data collected by Azure Monitor fits into one of two fundamental types, **metrics** and **logs**. Metrics are numerical values that describe some aspect of a system at a particular point in time. They are lightweight and capable of supporting near real-time scenarios. Logs contain different kinds of data organized into records with different sets of properties for each type. Telemetry such as events and traces are stored as logs in addition to performance data so that it can all be combined for analysis.

Log data collected by Azure Monitor is stored in Log Analytics which includes a rich query language to quickly retrieve, consolidate, and analyze collected data. You can create and test queries using the Log Analytics page in the Azure portal and then either directly analyze the data using these tools or save queries for use with visualizations or alert rules.

What data does Azure Monitor collect?

Azure Monitor can collect data from a variety of sources. You can think of monitoring data for your applications in tiers ranging from your application, any operating system and services it relies on, down to the platform itself. Azure Monitor collects data from each of the following tiers:

- **Application monitoring data:** Data about the performance and functionality of the code you have written, regardless of its platform.
- **Guest OS monitoring data:** Data about the operating system on which your application is running. This could be running in Azure, another cloud, or on-premises.
- **Azure resource monitoring data:** Data about the operation of an Azure resource.
- **Azure subscription monitoring data:** Data about the operation and management of an Azure subscription, as well as data about the health and operation of Azure itself.
- **Azure tenant monitoring data:** Data about the operation of tenant-level Azure services, such as Azure Active Directory.

As soon as you create an Azure subscription and start adding resources such as virtual machines and web apps, Azure Monitor starts collecting data. Activity Logs record when resources are created or modified. Metrics tell you how the resource is performing and the resources that it's consuming.

Extend the data you're collecting into the actual operation of the resources by enabling diagnostics and adding an agent to compute resources. This will collect telemetry for the internal operation of the resource and allow you to configure different data sources to collect logs and metrics from Windows and Linux guest operating system.

Add an instrumentation package to your application, to enable Application Insights to collect detailed information about your application including page views, application requests, and exceptions. Further verify the availability of your application by configuring an availability test to simulate user traffic.

Insights

Monitoring data is only useful if it can increase your visibility into the operation of your computing environment. Azure Monitor includes several features and tools that provide valuable insights into your applications and other resources that they depend on. Monitoring solutions and features such as Application Insights and Container Insights provide deep insights into different aspects of your application and specific Azure services.

Application Insights

Application Insights monitors the availability, performance, and usage of your web applications whether they're hosted in the cloud or on-premises. It leverages the powerful data analysis platform in Azure Monitor to provide you with deep insights into your application's operations and diagnose errors without waiting for a user to report them. Application Insights includes connection points to a variety of development tools and integrates with Visual Studio to support your DevOps processes.

Azure Monitor for containers

Azure Monitor for containers is a feature designed to monitor the performance of container workloads deployed to managed Kubernetes clusters hosted on Azure Kubernetes Service (AKS). It gives you performance visibility by collecting memory and processor metrics from controllers, nodes, and containers that are available in Kubernetes through the Metrics API. Container logs are also collected. After you enable monitoring from Kubernetes clusters, these metrics and logs are automatically collected for you through a containerized version of the Log Analytics agent for Linux.

Azure Monitor for VMs

Azure Monitor VM insights monitors your Azure virtual machines (VM) at scale by analyzing the performance and health of your Windows and Linux VMs, including their different processes and interconnected dependencies on other resources and external processes. The solution includes support for monitoring performance and application dependencies for VMs hosted on-premises or another cloud provider.

Monitoring solutions

Monitoring solutions in Azure Monitor are packaged sets of logic that provide insights for a particular application or service. They include logic for collecting monitoring data for the application or service, queries to analyze that data, and views for visualization. Monitoring solutions are available from Microsoft and partners to provide monitoring for various Azure services and other applications.

Responding to critical situations

In addition to allowing you to interactively analyze monitoring data, an effective monitoring solution must be able to proactively respond to critical conditions identified in the data that it collects. This could be sending a text or mail to an administrator responsible for investigating an issue. Or you could launch an automated process that attempts to correct an error condition.

Alerts

Alerts in Azure Monitor proactively notify you of critical conditions and potentially attempt to take corrective action. Alert rules based on metrics provide near real time alerting based on numeric values, while rules based on logs allow for complex logic across data from multiple sources.

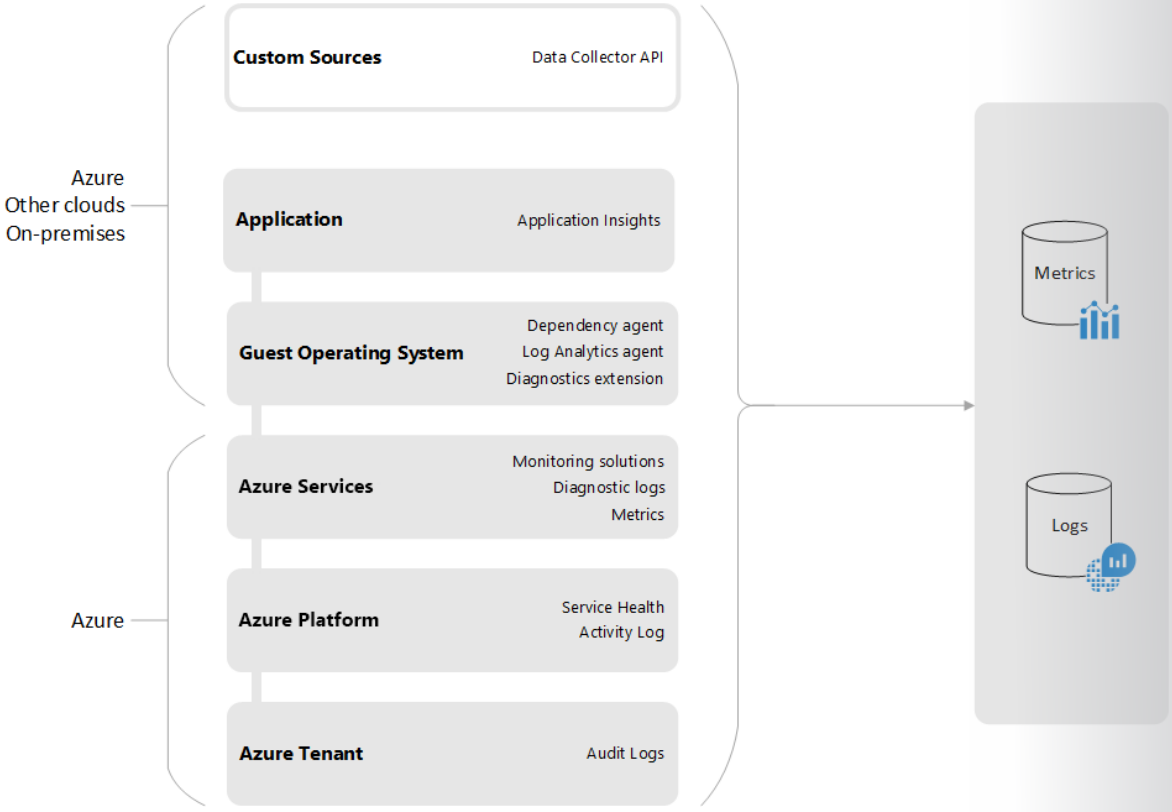
Alert rules in Azure Monitor use action groups, which contain unique sets of recipients and actions that can be shared across multiple rules. Based on your requirements, action groups can perform such actions as using webhooks to have alerts start external actions or to integrate with your ITSM tools.

Autoscale

Autoscale allows you to have the right amount of resources running to handle the load on your application. It allows you to create rules that use metrics collected by Azure Monitor to determine when to automatically add resources to handle increases in load and also save money by removing resources that are sitting idle. You specify a minimum and maximum number of instances and the logic for when to increase or decrease resources.

Sources of data in Azure Monitor

Monitoring data in Azure comes from a variety of sources that can be organized into tiers, the highest tiers being your application and any operating systems and the lower tiers being components of Azure platform. This is illustrated in the following diagram with descriptions below.



- **Azure Tenant:** Telemetry related to your Azure tenant is collected from tenant-wide services such as Azure Active Directory.
- **Azure platform:** Telemetry related to the health and operation of Azure itself includes data about the operation and management of your Azure subscription. It includes service health data stored in the Azure Activity log and audit logs from Azure Active Directory.
- **Guest operating system:** Compute resources in Azure, in other clouds, and on-premises have a guest operating system to monitor. With the installation of one or more agents, you can gather telemetry from the guest into the same monitoring tools as the Azure services themselves.
- **Applications:** In addition to telemetry that your application may write to the guest operating system, detailed application monitoring is done with Application Insights. Application Insights can collect data from applications running on a variety of platforms. The application can be running in Azure, another cloud, or on-premises.
- **Custom sources:** Azure Monitor can collect log data from any REST client using the Data Collector API. This allows you to create custom monitoring scenarios and extend monitoring to resources that don't expose telemetry through other sources.

Application Insights

Application Insights is an extensible Application Performance Management (APM) service for web developers on multiple platforms. Use it to monitor your live web application. It will automatically detect performance anomalies. It includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app.

What does Application Insights monitor?

Application Insights is aimed at the development team, to help you understand how your app is performing and how it's being used. It monitors:

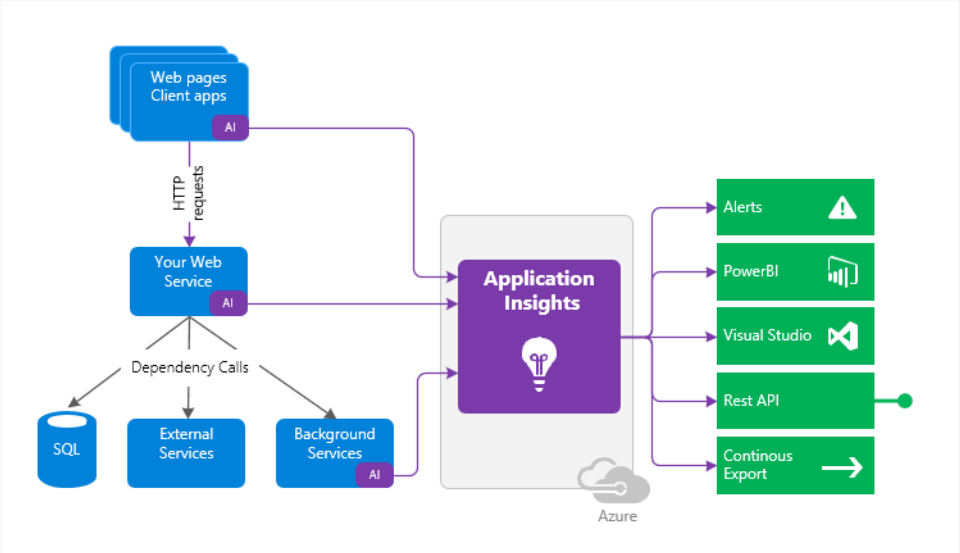
- **Request rates, response times, and failure rates** - Find out which pages are most popular, at what times of day, and where your users are. See which pages perform best. If your response times and failure rates go high when there are more requests, then perhaps you have a resourcing problem.
- **Dependency rates, response times, and failure rates** - Find out whether external services are slowing you down.
- **Exceptions** - Analyse the aggregated statistics, or pick specific instances and drill into the stack trace and related requests. Both server and browser exceptions are reported.
- **Page views and load performance** - reported by your users' browsers.
- **AJAX calls from web pages** - rates, response times, and failure rates.
- **User and session counts.**
- **Performance counters** from your Windows or Linux server machines, such as CPU, memory, and network usage.
- **Host diagnostics** from Docker or Azure.
- **Diagnostic trace logs** from your app - so that you can correlate trace events with requests.
- **Custom events and metrics** that you write yourself in the client or server code, to track business events such as items sold or games won.

How Application Insights works

You install a small instrumentation package in your application, and set up an Application Insights resource in the Microsoft Azure portal. The instrumentation monitors your app and sends telemetry data to the portal. (The application can run anywhere - it doesn't have to be hosted in Azure.)

Note: The impact on your app's performance is very small. Tracking calls are non-blocking, and are batched and sent in a separate thread.

You can instrument not only the web service application, but also any background components, and the JavaScript in the web pages themselves.



In addition, you can pull in telemetry from the host environments such as performance counters, Azure diagnostics, or Docker logs. You can also set up web tests that periodically send synthetic requests to your web service.

All these telemetry streams are integrated in the Azure portal, where you can apply powerful analytic and search tools to the raw data.

Overview of alerts in Microsoft Azure

This lesson describes what alerts are, their benefits, and how to get started using them.

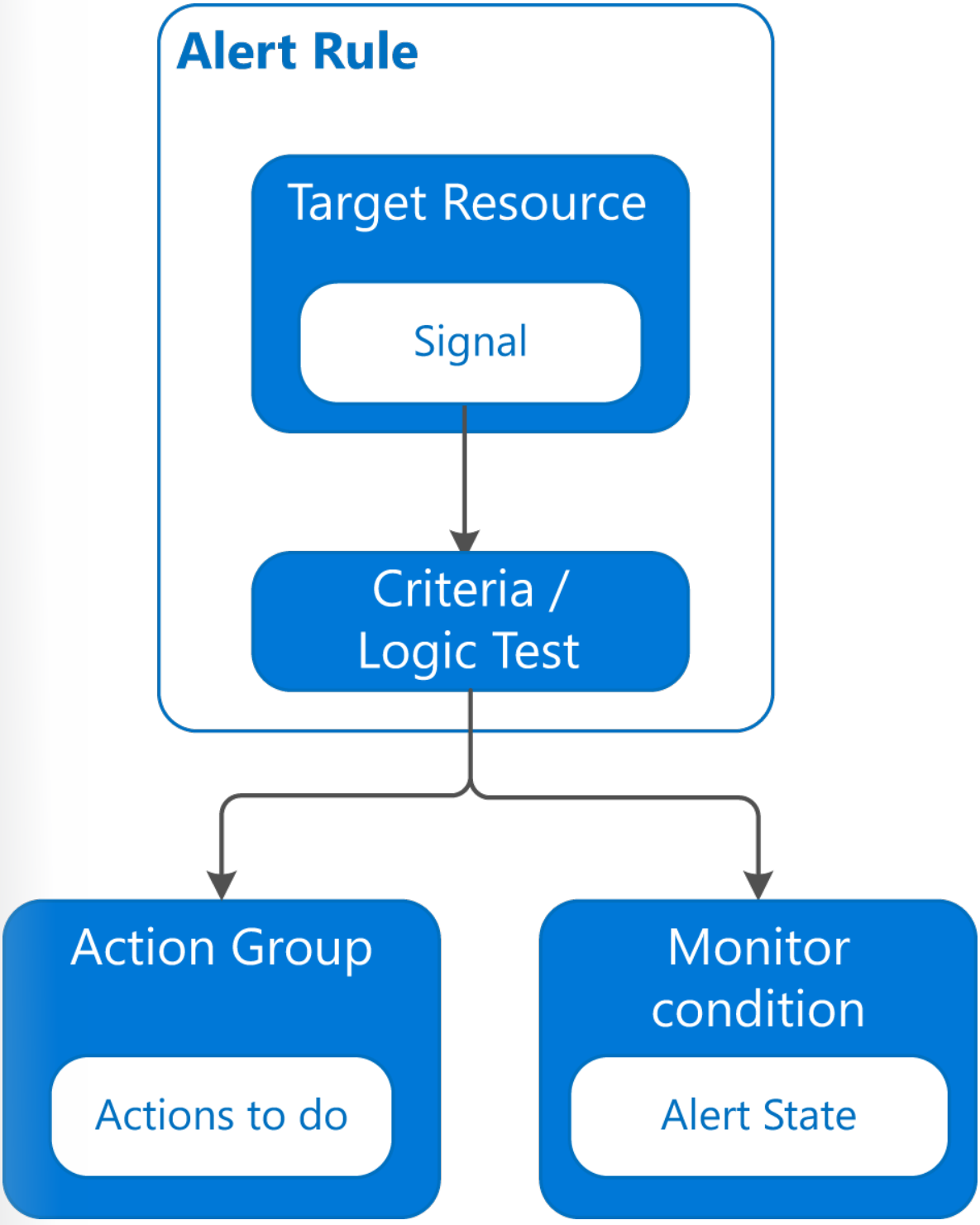
What are alerts in Microsoft Azure?

Alerts proactively notify you when important conditions are found in your monitoring data. They allow you to identify and address issues before the users of your system notice them.

Note: This lesson discusses the unified alert experience in Azure Monitor, which now includes Log Analytics and Application Insights. The previous alert experience and alert types are called **classic alerts**. You can view this older experience and older alert type by clicking on **View classic alerts** at the top of the alert page.

Overview

The diagram below represents the flow of alerts.



Alert rules are separated from alerts and the action that are taken when an alert fires.

Alert rule - The alert rule captures the target and criteria for alerting. The alert rule can be in an enabled or a disabled state. Alerts only fire when enabled.

The key attributes of an alert rule are:

- **Target Resource** - Defines the scope and signals available for alerting. A target can be any Azure resource. Example targets: a virtual machine, a storage account, a virtual machine scale set, a Log Analytics workspace, or an Application Insights resource. For certain resources (like Virtual Machines), you can specify multiple resources as the target of the alert rule.
- **Signal** - Signals are emitted by the target resource and can be of several types. Metric, Activity log, Application Insights, and Log.
- **Criteria** - Criteria is combination of Signal and Logic applied on a Target resource. Examples:
  - Percentage CPU > 70%
  - Server Response Time > 4 ms
  - Result count of a log query > 100
- **Alert Name** – A specific name for the alert rule configured by the user
- **Alert Description** – A description for the alert rule configured by the user
- **Severity** – The severity of the alert once the criteria specified in the alert rule is met. Severity can range from 0 to 4.
- **Action** - A specific action taken when the alert is fired. For more information, see Action Groups.

What you can alert on

You can alert on metrics and logs. These include but are not limited to:

- Metric values
- Log search queries
- Activity Log events
- Health of the underlying Azure platform
- Tests for web site availability

Manage alerts

You can set the state of an alert to specify where it is in the resolution process. When the criteria specified in the alert rule is met, an alert is created or fired, it has a status of New. You can change the status when you acknowledge an alert and when you close it. All state changes are stored in the history of the alert.

The following alert states are supported.

State	Description
New	The issue has just been detected and has not yet been reviewed.
Acknowledged	An administrator has reviewed the alert and started working on it.
Closed	The issue has been resolved. After an alert has been closed, you can reopen it by changing it to another state.

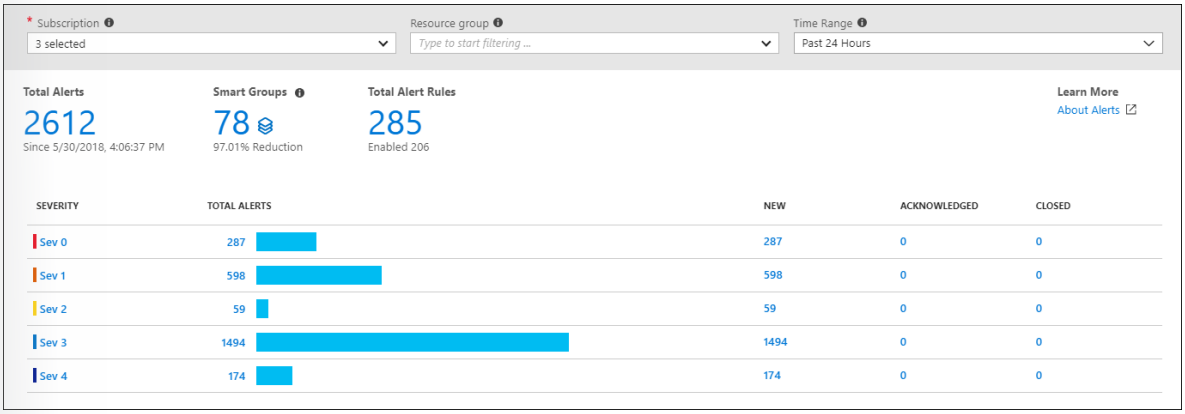
Alert state is different and independent of the **monitor condition**. Alert state is set by the user. Monitor condition is set by the system. When an alert fires, the alert's monitor condition is set to fired. When the

underlying condition that caused the alert to fire clears, the monitor condition is set to resolved. The alert state isn't changed until the user changes it.

Alerts experience

The default Alerts page provides a summary of alerts that are created within a particular time window. It displays the total alerts for each severity with columns that identify the total number of alerts in each state for each severity. Select any of the severities to open the All Alerts page filtered by that severity.

It does not show or track older classic alerts. You can change the subscriptions or filter parameters to update the page.



Create an alert rule

Alerts can be authored in a consistent manner regardless of the monitoring service or signal type. All fired alerts and related details are available in single page.

You create a new alert rule with the following three steps:

1. Pick the *target* for the alert.
2. Select the *signal* from the available signals for the target.
3. Specify the *logic* to be applied to data from the signal.

This simplified authoring process no longer requires you to know the monitoring source or signals that are supported before selecting an Azure resource. The list of available signals is automatically filtered based on the target resource that you select. Also based on that target, you are guided through defining the logic of the alert rule automatically.

Review questions

Module 1 review questions

Azure Monitor data types

All data collected by Azure Monitor fits into one of two fundamental types, **metrics** and **logs**. What kinds of information is collected for each fundamental type?

> Click to see suggested answer

- Metrics are numerical values that describe some aspect of a system at a particular point in time. They are lightweight and capable of supporting near real-time scenarios.
- Logs contain different kinds of data organized into records with different sets of properties for each type. Telemetry such as events and traces are stored as logs in addition to performance data so that it can all be combined for analysis.

Azure Monitor scope

Is Azure Monitor limited to monitoring only apps and services hosted in Azure?

> Click to see suggested answer

Compute resources in Azure, in other clouds, and on-premises have a guest operating system to monitor. With the installation of one or more agents, you can gather telemetry from the guest into the same monitoring tools as the Azure services themselves.

Monitored metrics

Application Insights is aimed at the development team to help you understand how your app is performing and how it's being used. It monitors many metrics, how many can you think of?

> Click to see suggested answer

- Request rates, response times, and failure rates - Find out which pages are most popular, what times of day are most popular, and where your users are. See which pages perform the best. If your response times and failure rates go high when there are more requests, perhaps you have a resourcing problem.
- Dependency rates, response times, and failure rates - Find out whether external services are slowing you down.
- Exceptions - Analyze the aggregated statistics, or pick specific instances and drill into the stack trace and related requests. Both server and browser exceptions are reported.
- Page views and load performance reported by your users' browsers.
- Asynchronous JavaScript And XML (AJAX) calls from webpages - Rates, response times, and failure rates.
- User and session counts.

- Performance counters from your Windows Server or Linux server machines, such as those for CPU, memory, and network usage.
- Host diagnostics from Docker or Azure.
- Diagnostic trace logs from your app so that you can correlate trace events with requests.
- Custom events and metrics that you write yourself in the client or server code to track business events, such as the number of items sold or games won.

Module 2 Develop code to support scalability of apps and services

Implement autoscale

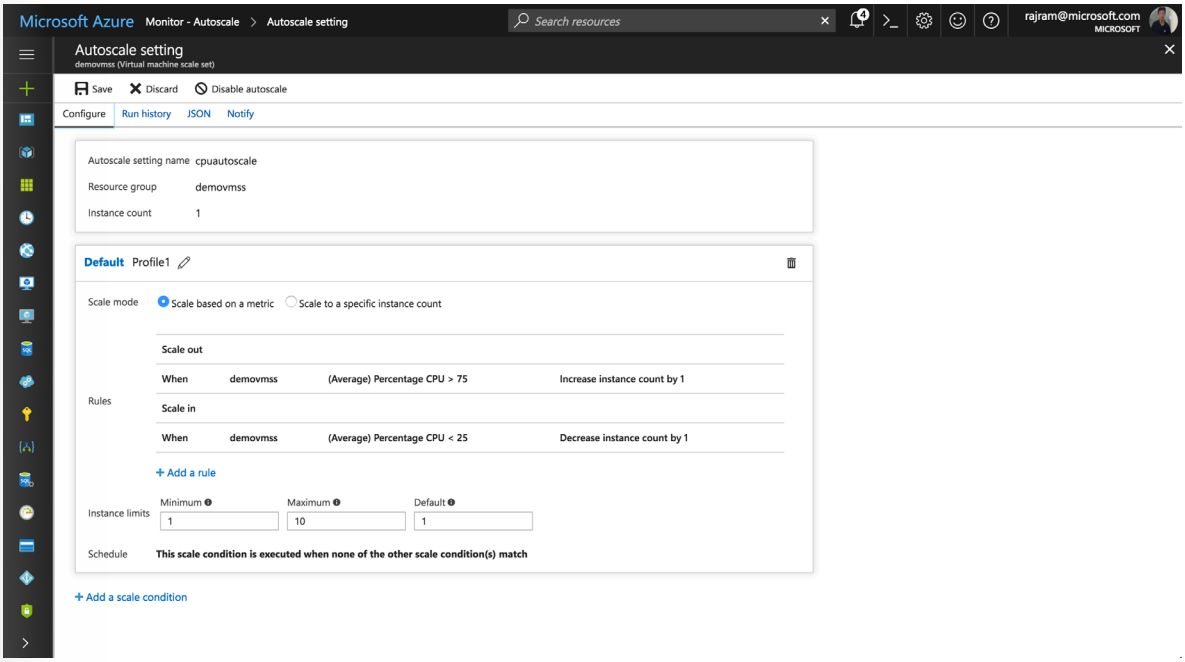
Common autoscale patterns

Note: Azure Monitor autoscale currently applies only to Virtual Machine Scale Sets, Cloud Services, App Service - Web Apps, and API Management services.

Scale based on CPU

You have a web app (/VMSS/cloud service role) and

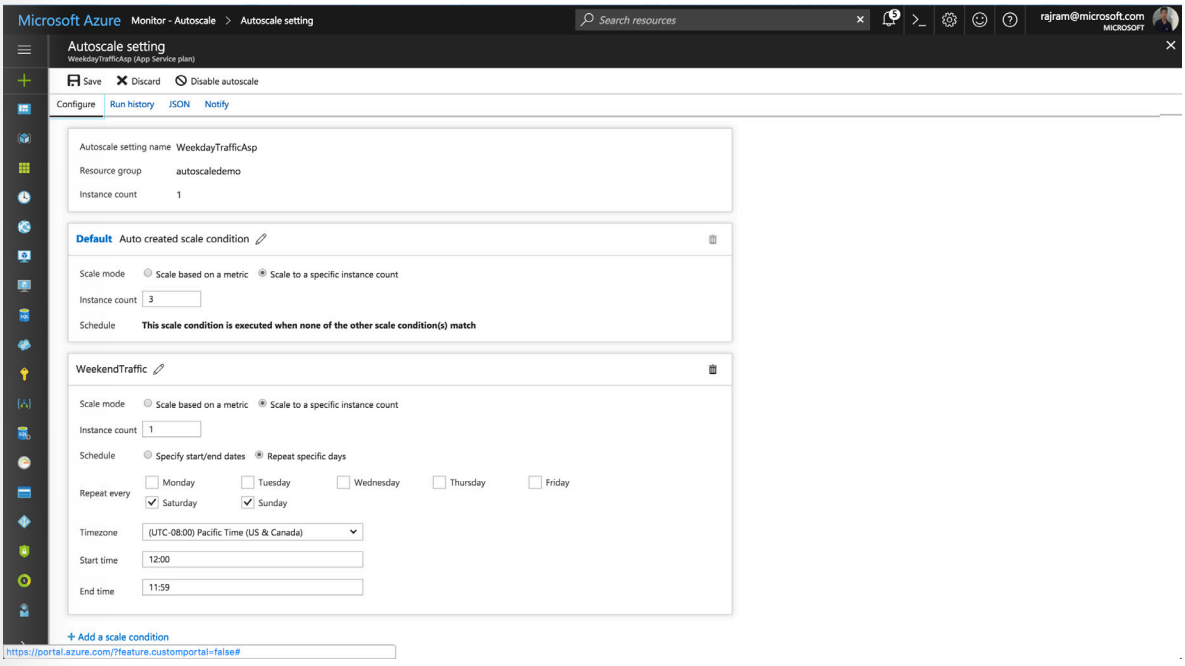
- You want to scale out/scale in based on CPU.
- Additionally, you want to ensure there is a minimum number of instances.
- Also, you want to ensure that you set a maximum limit to the number of instances you can scale to.



Scale differently on weekdays vs weekends

You have a web app (/VMSS/cloud service role) and

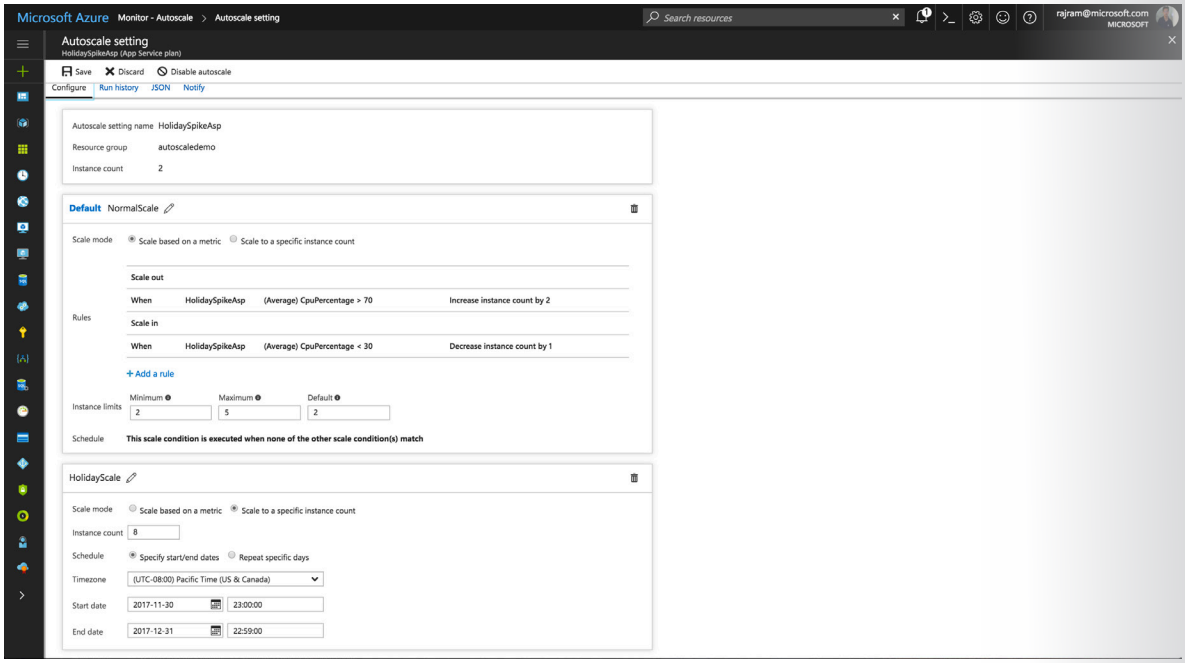
- You want 3 instances by default (on weekdays)
- You don't expect traffic on weekends and hence you want to scale down to 1 instance on weekends.



Scale differently during holidays

You have a web app (/VMSS/cloud service role) and

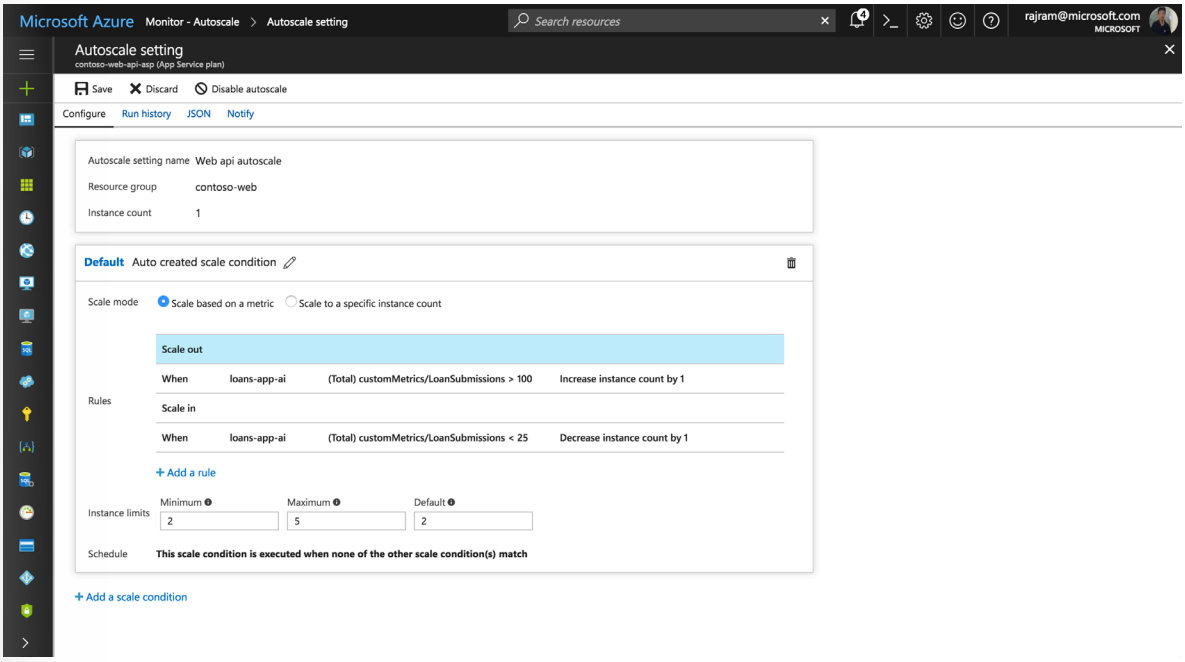
- You want to scale up/down based on CPU usage by default
- However, during holiday season (or specific days that are important for your business) you want to override the defaults and have more capacity at your disposal.



Scale based on custom metric

You have a web front end and a API tier that communicates with the backend.

- You want to scale the API tier based on custom events in the front end (example: You want to scale your checkout process based on the number of items in the shopping cart)



Understand Autoscale settings

Autoscale settings help ensure that you have the right amount of resources running to handle the fluctuating load of your application. You can configure Autoscale settings to be triggered based on metrics that indicate load or performance, or triggered at a scheduled date and time. This article takes a detailed look at the anatomy of an Autoscale setting. The article begins with the schema and properties of a setting, and then walks through the different profile types that can be configured. Finally, the article discusses how the Autoscale feature in Azure evaluates which profile to execute at any given time.

Autoscale setting schema

To illustrate the Autoscale setting schema, the following Autoscale setting is used. It is important to note that this Autoscale setting has:

- One profile.
- Two metric rules in this profile: one for scale out, and one for scale in.
    - The scale-out rule is triggered when the virtual machine scale set's average percentage CPU metric is greater than 85 percent for the past 10 minutes.
    - The scale-in rule is triggered when the virtual machine scale set's average is less than 60 percent for the past minute.

```
{
  "id": "/subscriptions/s1/resourceGroups/rg1/providers/microsoft.insights/
autoscalesettings/setting1",
  "name": "setting1",
  "type": "Microsoft.Insights/autoscaleSettings",
  "location": "East US",
  "properties": {
```

```
    "enabled": true,
    "targetResourceUri": "/subscriptions/s1/resourceGroups/rg1/providers/
Microsoft.Compute/virtualMachineScaleSets/vmss1",
    "profiles": [
      {
        "name": "mainProfile",
        "capacity": {
          "minimum": "1",
          "maximum": "4",
          "default": "1"
        },
        "rules": [
          {
            "metricTrigger": {
              "metricName": "Percentage CPU",
              "metricResourceUri": "/subscriptions/s1/resourceGroups/rg1/
providers/Microsoft.Compute/virtualMachineScaleSets/vmss1",
              "timeGrain": "PT1M",
              "statistic": "Average",
              "timeWindow": "PT10M",
              "timeAggregation": "Average",
              "operator": "GreaterThan",
              "threshold": 85
            },
            "scaleAction": {
              "direction": "Increase",
              "type": "ChangeCount",
              "value": "1",
              "cooldown": "PT5M"
            }
          },
    {
            "metricTrigger": {
              "metricName": "Percentage CPU",
              "metricResourceUri": "/subscriptions/s1/resourceGroups/rg1/
providers/Microsoft.Compute/virtualMachineScaleSets/vmss1",
              "timeGrain": "PT1M",
              "statistic": "Average",
              "timeWindow": "PT10M",
              "timeAggregation": "Average",
              "operator": "LessThan",
              "threshold": 60
            },
            "scaleAction": {
              "direction": "Decrease",
              "type": "ChangeCount",
              "value": "1",
              "cooldown": "PT5M"
            }
          }
        ]
```

```
            }
        ]
    }
}
```

Section	Element name	Description
Setting	ID	The Autoscale setting's resource ID. Autoscale settings are an Azure Resource Manager resource.
Setting	name	The Autoscale setting name.
Setting	location	The location of the Autoscale setting. This location can be different from the location of the resource being scaled.
properties	targetResourceUri	The resource ID of the resource being scaled. You can only have one Autoscale setting per resource.
properties	profiles	An Autoscale setting is composed of one or more profiles. Each time the Autoscale engine runs, it executes one profile.
profile	name	The name of the profile. You can choose any name that helps you identify the profile.
profile	Capacity.maximum	The maximum capacity allowed. It ensures that Autoscale, when executing this profile, does not scale your resource above this number.
profile	Capacity.minimum	The minimum capacity allowed. It ensures that Autoscale, when executing this profile, does not scale your resource below this number.

Section	Element name	Description
profile	Capacity.default	If there is a problem reading the resource metric (in this case, the CPU of "vmss1"), and the current capacity is below the default, Autoscale scales out to the default. This is to ensure the availability of the resource. If the current capacity is already higher than the default capacity, Autoscale does not scale in.
profile	rules	Autoscale automatically scales between the maximum and minimum capacities, by using the rules in the profile. You can have multiple rules in a profile. Typically there are two rules: one to determine when to scale out, and the other to determine when to scale in.
rule	metricTrigger	Defines the metric condition of the rule.
metricTrigger	metricName	The name of the metric.
metricTrigger	metricResourceUri	The resource ID of the resource that emits the metric. In most cases, it is the same as the resource being scaled. In some cases, it can be different. For example, you can scale a virtual machine scale set based on the number of messages in a storage queue.
metricTrigger	timeGrain	The metric sampling duration. For example, **TimeGrain = "PT1M"** means that the metrics should be aggregated every 1 minute, by using the aggregation method specified in the statistic element.

Section	Element name	Description
metricTrigger	statistic	The aggregation method within the timeGrain period. For example, **statistic = "Average"** and **timeGrain = "PT1M"** means that the metrics should be aggregated every 1 minute, by taking the average. This property dictates how the metric is sampled.
metricTrigger	timeWindow	The amount of time to look back for metrics. For example, **timeWindow = "PT10M"** means that every time Autoscale runs, it queries metrics for the past 10 minutes. The time window allows your metrics to be normalized, and avoids reacting to transient spikes.
metricTrigger	timeAggregation	The aggregation method used to aggregate the sampled metrics. For example, **TimeAggregation = "Average"** should aggregate the sampled metrics by taking the average. In the preceding case, take the ten 1-minute samples, and average them.
rule	scaleAction	The action to take when the metricTrigger of the rule is triggered.
scaleAction	direction	"Increase" to scale out, or "Decrease" to scale in.
scaleAction	value	How much to increase or decrease the capacity of the resource.

Section	Element name	Description
scaleAction	cooldown	The amount of time to wait after a scale operation before scaling again. For example, if **cooldown = "PT10M"**, Autoscale does not attempt to scale again for another 10 minutes. The cooldown is to allow the metrics to stabilize after the addition or removal of instances.

Autoscale profiles

There are three types of Autoscale profiles:

- **Regular profile:** The most common profile. If you don't need to scale your resource based on the day of the week, or on a particular day, you can use a regular profile. This profile can then be configured with metric rules that dictate when to scale out and when to scale in. You should only have one regular profile defined.
- The example profile used earlier in this article is an example of a regular profile. Note that it is also possible to set a profile to scale to a static instance count for your resource.
- **Fixed date profile:** This profile is for special cases. For example, let's say you have an important event coming up on December 26, 2017 (PST). You want the minimum and maximum capacities of your resource to be different on that day, but still scale on the same metrics. In this case, you should add a fixed date profile to your setting's list of profiles. The profile is configured to run only on the event's day. For any other day, Autoscale uses the regular profile.

```
"profiles": [{
"name": " regularProfile",
"capacity": {
...
},
"rules": [{
...
},
{
...
}]
},
{
"name": "eventProfile",
"capacity": {
...
},
"rules": [{
...
}, {
...
}],
"fixedDate": {
```

```
    "timeZone": "Pacific Standard Time",
        "start": "2017-12-26T00:00:00",
        "end": "2017-12-26T23:59:00"
}}
]
```

- **Recurrence profile:** This type of profile enables you to ensure that this profile is always used on a particular day of the week. Recurrence profiles only have a start time. They run until the next recurrence profile or fixed date profile is set to start. An Autoscale setting with only one recurrence profile runs that profile, even if there is a regular profile defined in the same setting. The following example illustrates a way this profile is used:

Weekdays vs. weekends

Let's say that on weekends, you want your maximum capacity to be 4. On weekdays, because you expect more load, you want your maximum capacity to be 10. In this case, your setting would contain two recurrence profiles, one to run on weekends and the other on weekdays. The setting looks like this:

```
"profiles": [
{
"name": "weekdayProfile",
"capacity": {
    ...
},
"rules": [{
    ...
}],
"recurrence": {
    "frequency": "Week",
    "schedule": {
        "timeZone": "Pacific Standard Time",
        "days": [
            "Monday"
        ],
        "hours": [
            0
        ],
        "minutes": [
            0
        ]
    }
}}
},
{
"name": "weekendProfile",
"capacity": {
    ...
},
"rules": [{
    ...
}]
```

```
"recurrence": {
    "frequency": "Week",
    "schedule": {
        "timeZone": "Pacific Standard Time",
        "days": [
            "Saturday"
        ],
        "hours": [
            0
        ],
        "minutes": [
            0
        ]
    }
}
}]
```

The preceding setting shows that each recurrence profile has a schedule. This schedule determines when the profile starts running. The profile stops when it's time to run another profile.

For example, in the preceding setting, "weekdayProfile" is set to start on Monday at 12:00 AM. That means this profile starts running on Monday at 12:00 AM. It continues until Saturday at 12:00 AM, when "weekendProfile" is scheduled to start running.

Autoscale evaluation

Given that Autoscale settings can have multiple profiles, and each profile can have multiple metric rules, it is important to understand how an Autoscale setting is evaluated. Each time the Autoscale job runs, it begins by choosing the profile that is applicable. Then Autoscale evaluates the minimum and maximum values, and any metric rules in the profile, and decides if a scale action is necessary.

Which profile will Autoscale pick?

Autoscale uses the following sequence to pick the profile:

1. It first looks for any fixed date profile that is configured to run now. If there is, Autoscale runs it. If there are multiple fixed date profiles that are supposed to run, Autoscale selects the first one.
2. If there are no fixed date profiles, Autoscale looks at recurrence profiles. If a recurrence profile is found, it runs it.
3. If there are no fixed date or recurrence profiles, Autoscale runs the regular profile.

How does Autoscale evaluate multiple rules?

After Autoscale determines which profile to run, it evaluates all the scale-out rules in the profile (these are rules with **direction = "Increase"**).

If one or more scale-out rules are triggered, Autoscale calculates the new capacity determined by the scaleAction of each of those rules. Then it scales out to the maximum of those capacities, to ensure service availability.

For example, let's say there is a virtual machine scale set with a current capacity of 10. There are two scale-out rules: one that increases capacity by 10 percent, and one that increases capacity by 3 counts.

The first rule would result in a new capacity of 11, and the second rule would result in a capacity of 13. To ensure service availability, Autoscale chooses the action that results in the maximum capacity, so the second rule is chosen.

If no scale-out rules are triggered, Autoscale evaluates all the scale-in rules (rules with **direction = "Decrease"**). Autoscale only takes a scale-in action if all of the scale-in rules are triggered.

Autoscale calculates the new capacity determined by the scaleAction of each of those rules. Then it chooses the scale action that results in the maximum of those capacities to ensure service availability.

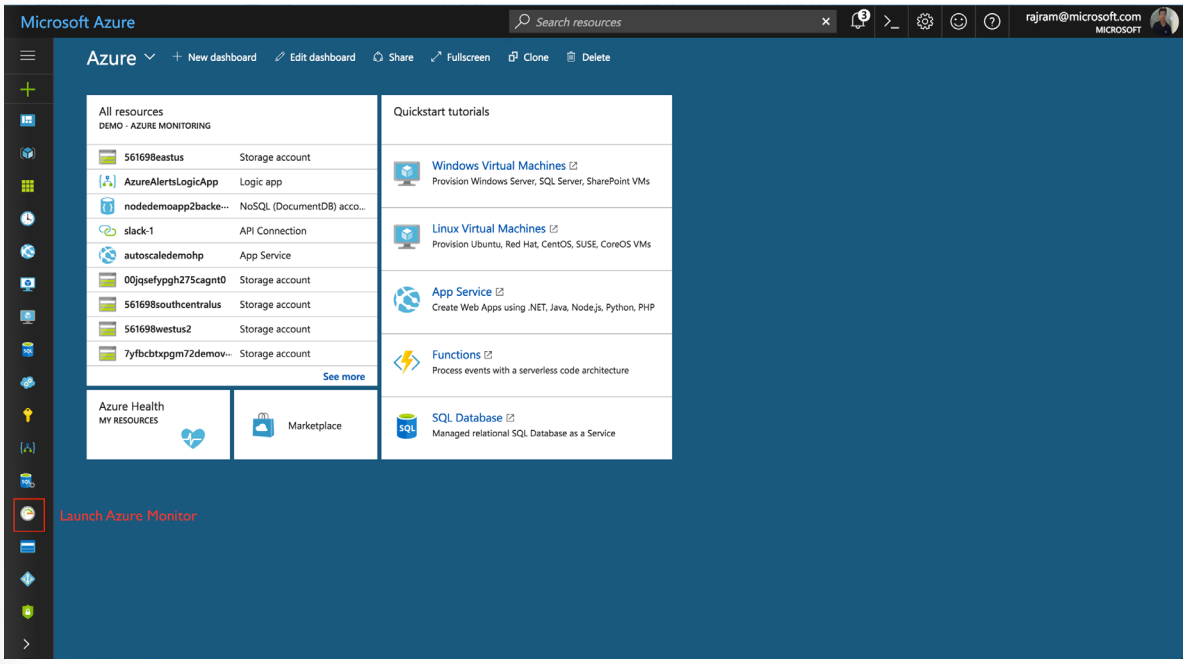
For example, let's say there is a virtual machine scale set with a current capacity of 10. There are two scale-in rules: one that decreases capacity by 50 percent, and one that decreases capacity by 3 counts. The first rule would result in a new capacity of 5, and the second rule would result in a capacity of 7. To ensure service availability, Autoscale chooses the action that results in the maximum capacity, so the second rule is chosen.

How to set autoscale by using a custom metric

Getting started

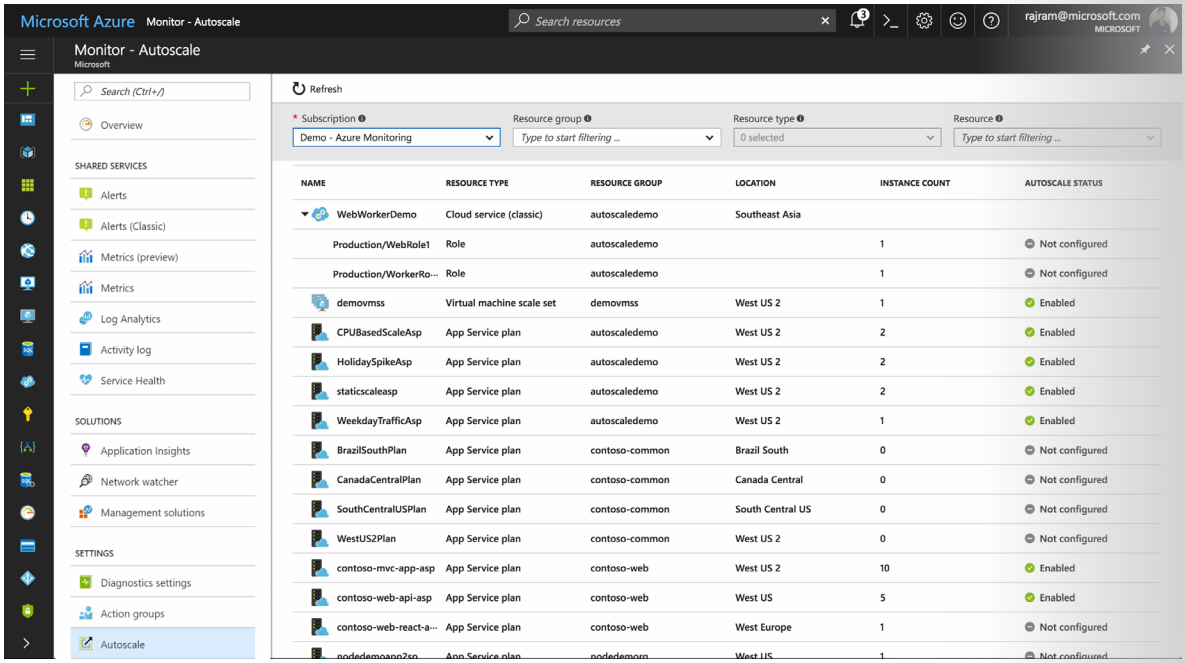
This lesson assumes that you have a web app with application insights configured. If you don't have one already, you can **set up Application Insights for your ASP.NET website**[1].

1. Open Azure portal
2. Click on **Azure Monitor** icon in the left navigation pane.

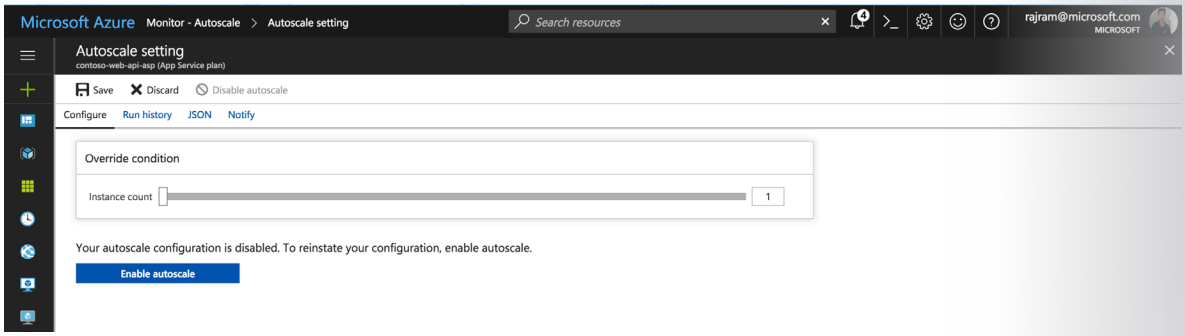


1 https://docs.microsoft.com/azure/application-insights/app-insights-asp-net

3. Click on **Autoscale** setting to view all the resources for which auto scale is applicable, along with its current autoscale status

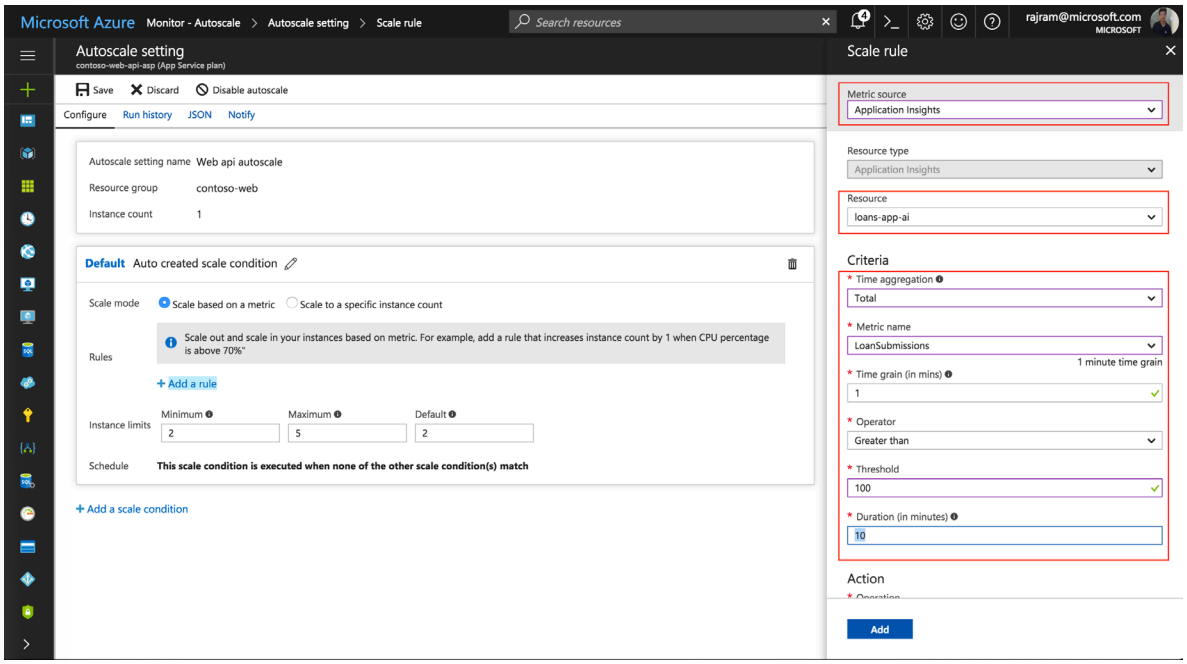


4. Open the **Autoscale** blade in Azure Monitor and select a resource you want to scale
5. **Note:** The steps below use an app service plan associated with a web app that has App Insights configured.
6. In the scale setting blade for the resource, notice that the current instance count is Click on **Enable autoscale**.

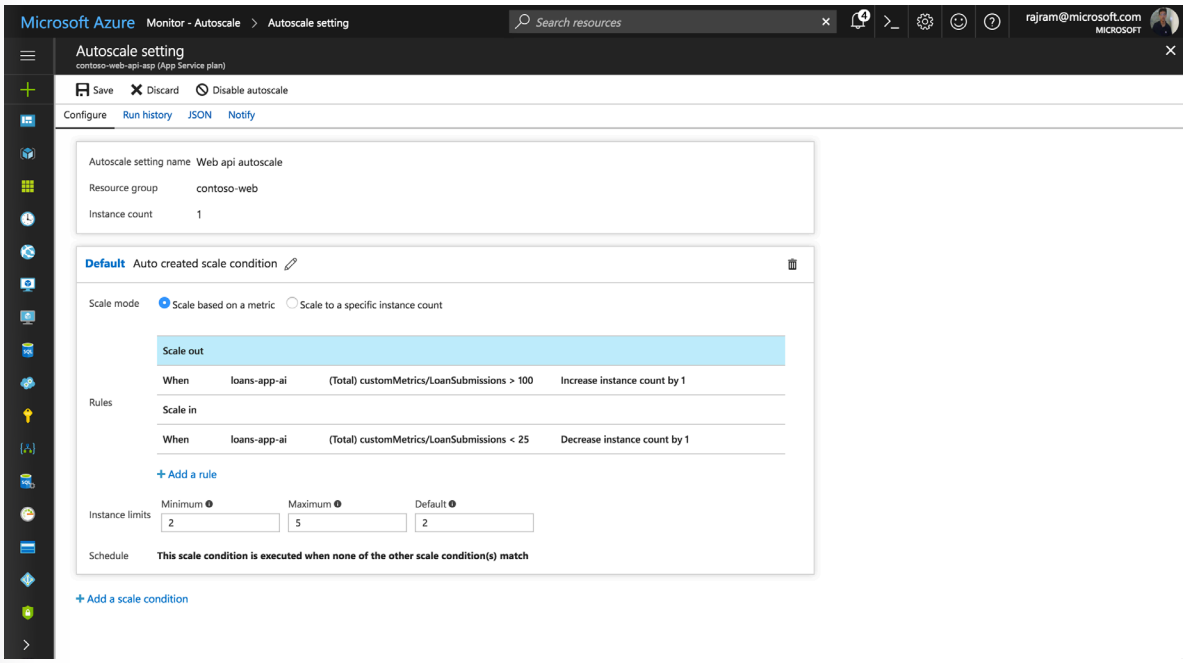


7. Provide a name for the scale setting, and the click on **Add a rule**. Notice the scale rule options that opens as a context pane in the right hand side. By default, it sets the option to scale your instance count by 1 if the CPU percentage of the resource exceeds 70%. Change the metric source at the top to **Application Insights**, select the app insights resource in the **Resource** dropdown and then select

the custom metric based on which you want to scale.



8. Similar to the step above, add a scale rule that will scale in and decrease the scale count by 1 if the custom metric is below a threshold.



9. Set the you instance limits. For example, if you want to scale between 2-5 instances depending on the custom metric fluctuations, set **Minimum** to '2', **Maximum** to '5' and **Default** to '2'
10. **Note:** In case there is a problem reading the resource metrics and the current capacity is below the default capacity, then to ensure the availability of the resource, Autoscale will scale out to the default value. If the current capacity is already higher than default capacity, Autoscale will not scale in.
11. Click on **Save**

Congratulations. You now successfully created your scale setting to auto scale your web app based on a custom metric.

Best practices for Autoscale

Autoscale concepts

- A resource can have only one autoscale setting
- An autoscale setting can have one or more profiles and each profile can have one or more autoscale rules.
- An autoscale setting scales instances horizontally, which is *out* by increasing the instances and *in* by decreasing the number of instances. An autoscale setting has a maximum, minimum, and default value of instances.
- An autoscale job always reads the associated metric to scale by, checking if it has crossed the configured threshold for scale-out or scale-in. You can view a list of metrics that autoscale can scale by at **Azure Monitor autoscaling common metrics**[2].
- All thresholds are calculated at an instance level. For example, "scale out by one instance when average CPU > 80% when instance count is 2", means scale-out when the average CPU across all instances is greater than 80%.
- All autoscale failures are logged to the Activity Log. You can then configure an activity log alert so that you can be notified via email, SMS, or webhooks whenever there is an autoscale failure.
- Similarly, all successful scale actions are posted to the Activity Log. You can then configure an activity log alert so that you can be notified via email, SMS, or webhooks whenever there is a successful autoscale action. You can also configure email or webhook notifications to get notified for successful scale actions via the notifications tab on the autoscale setting.

Autoscale best practices

Use the following best practices as you use autoscale.

Ensure the maximum and minimum values are different and have an adequate margin between them

If you have a setting that has minimum=2, maximum=2 and the current instance count is 2, no scale action can occur. Keep an adequate margin between the maximum and minimum instance counts, which are inclusive. Autoscale always scales between these limits.

Manual scaling is reset by autoscale min and max

If you manually update the instance count to a value above or below the maximum, the autoscale engine automatically scales back to the minimum (if below) or the maximum (if above). For example, you set the range between 3 and 6. If you have one running instance, the autoscale engine scales to three instances on its next run. Likewise, if you manually set the scale to eight instances, on the next run autoscale will scale it back to six instances on its next run. Manual scaling is temporary unless you reset the autoscale rules as well.

2 https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-common-metrics

Always use a scale-out and scale-in rule combination that performs an increase and decrease

If you use only one part of the combination, autoscale will only take action in a single direction (scale out, or in) until it reaches the maximum, or minimum istance counts of defined in the profile. This is not optimal, ideally you want your resource to scale up at times of high usage to ensure availability. Similarly, at times of low usage you want your resource to scale down, so you can realize cost savings.

Choose the appropriate statistic for your diagnostics metric

For diagnostics metrics, you can choose among *Average*, *Minimum*, *Maximum* and *Total* as a metric to scale by. The most common statistic is Average.

Choose the thresholds carefully for all metric types

We recommend carefully choosing different thresholds for scale-out and scale-in based on practical situations.

We *do not recommend* autoscale settings like the examples below with the same or very similar threshold values for out and in conditions:

- Increase instances by 1 count when Thread Count <= 600
- Decrease instances by 1 count when Thread Count >= 600

Let's look at an example of what can lead to a behavior that may seem confusing. Consider the following sequence.

1. Assume there are two instances to begin with and then the average number of threads per instance grows to 625.
2. Autoscale scales out adding a third instance.
3. Next, assume that the average thread count across instance falls to 575.
4. Before scaling down, autoscale tries to estimate what the final state will be if it scaled in. For example, 575 x 3 (current instance count) = 1,725 / 2 (final number of instances when scaled down) = 862.5 threads. This means autoscale would have to immediately scale-out again even after it scaled in, if the average thread count remains the same or even falls only a small amount. However, if it scaled up again, the whole process would repeat, leading to an infinite loop.
5. To avoid this situation (termed “flapping”), autoscale does not scale down at all. Instead, it skips and reevaluates the condition again the next time the service's job executes. This can confuse many people because autoscale wouldn't appear to work when the average thread count was 575.

Estimation during a scale-in is intended to avoid “flapping” situations, where scale-in and scale-out actions continually go back and forth. Keep this behavior in mind when you choose the same thresholds for scale-out and in.

We recommend choosing an adequate margin between the scale-out and in thresholds. As an example, consider the following better rule combination.

- Increase instances by 1 count when CPU% >= 80
- Decrease instances by 1 count when CPU% <= 60

In this case

1. Assume there are 2 instances to start with.

2. If the average CPU% across instances goes to 80, autoscale scales out adding a third instance.
3. Now assume that over time the CPU% falls to 60.
4. Autoscale's scale-in rule estimates the final state if it were to scale-in. For example, 60 x 3 (current instance count) = 180 / 2 (final number of instances when scaled down) = 90. So autoscale does not scale-in because it would have to scale-out again immediately. Instead, it skips scaling down.
5. The next time autoscale checks, the CPU continues to fall to 50. It estimates again - 50 x 3 instance = 150 / 2 instances = 75, which is below the scale-out threshold of 80, so it scales in successfully to 2 instances.

Considerations for scaling threshold values for special metrics

For special metrics such as Storage or Service Bus Queue length metric, the threshold is the average number of messages available per current number of instances. Carefully choose the threshold value for this metric.

Let's illustrate it with an example to ensure you understand the behavior better.

- Increase instances by 1 count when Storage Queue message count >= 50
- Decrease instances by 1 count when Storage Queue message count <= 10

Consider the following sequence:

1. There are two storage queue instances.
2. Messages keep coming and when you review the storage queue, the total count reads 50. You might assume that autoscale should start a scale-out action. However, note that it is still 50/2 = 25 messages per instance. So, scale-out does not occur. For the first scale-out to happen, the total message count in the storage queue should be 100.
3. Next, assume that the total message count reaches 100.
4. A 3rd storage queue instance is added due to a scale-out action. The next scale-out action will not happen until the total message count in the queue reaches 150 because 150/3 = 50.
5. Now the number of messages in the queue gets smaller. With three instances, the first scale-in action happens when the total messages in all queues add up to 30 because 30/3 = 10 messages per instance, which is the scale-in threshold.

Considerations for scaling when multiple rules are configured in a profile

There are cases where you may have to set multiple rules in a profile. The following set of autoscale rules are used by services use when multiple rules are set.

On *scale out*, autoscale runs if any rule is met. On *scale-in*, autoscale require all rules to be met.

To illustrate, assume that you have the following four autoscale rules:

- If CPU < 30 %, scale-in by 1
- If Memory < 50%, scale-in by 1
- If CPU > 75%, scale-out by 1
- If Memory > 75%, scale-out by 1

Then the follow occurs:

- If CPU is 76% and Memory is 50%, we scale-out.
- If CPU is 50% and Memory is 76% we scale-out.

On the other hand, if CPU is 25% and memory is 51% autoscale does not scale-in. In order to scale-in, CPU must be 29% and Memory 49%.

Always select a safe default instance count

The default instance count is important autoscale scales your service to that count when metrics are not available. Therefore, select a default instance count that's safe for your workloads.

Configure autoscale notifications

Autoscale will post to the Activity Log if any of the following conditions occur:

- Autoscale issues a scale operation
- Autoscale service successfully completes a scale action
- Autoscale service fails to take a scale action.
- Metrics are not available for autoscale service to make a scale decision.
- Metrics are available (recovery) again to make a scale decision.

You can also use an Activity Log alert to monitor the health of the autoscale engine. In addition to using activity log alerts, you can also configure email or webhook notifications to get notified for successful scale actions via the notifications tab on the autoscale setting.

Implement code that addresses singleton application instances

Querying resources using Azure CLI

The Azure Command-Line Interface (Azure CLI) is the Microsoft cross-platform command-line experience for managing Azure resources. You can use it in your browser with Azure Cloud Shell or install it on macOS, Linux, or Windows and run it from the command line. Azure CLI is optimized for managing and administering Azure resources from the command line and for building automation scripts that work against the Azure Resource Manager.

The Azure CLI uses the `--query` argument to execute a JMESPath query on the results of commands. JMESPath is a query language for JavaScript Object Notation (JSON) that gives you the ability to select and present data from Azure CLI output. These queries are executed on the JSON output before they perform any other display formatting. The `--query` argument is supported by all commands in the Azure CLI.

Many CLI commands will return more than one value. These commands always return a JSON array instead of a JSON document. Arrays can have their elements accessed by index, but there's never an order guarantee from the Azure CLI. To make the arrays easier to query, we can flatten them using the JMESPath `[]` operator.

In the following example, we use the `az vm list` command to query for a list of virtual machine (VM) instances:

```
az vm list
```

The query will return an array of large JSON objects for each VM in your subscription:

```
[
    {
          "availabilitySet": null,
          "diagnosticsProfile": null,
          "hardwareProfile": {
                "vmSize": "Standard_B1s"
          },
          "id": "/subscriptions/9103844d-1370-4716-b02b-69ce936865c6/
resourceGroups/VM/providers/Microsoft.Compute/virtualMachines/simple",
          "identity": null,
          "instanceView": null,
          "licenseType": null,
          "location": "eastus",
          "name": "simple",
          "networkProfile": {
                "networkInterfaces": [{
                      "id": "/subscriptions/9103844d-1370-4716-b02b-
69ce936865c6/resourceGroups/VM/providers/Microsoft.Network/networkInterfac-
es/simple159",
                      "primary": null,
                      "resourceGroup": "VM"
                }]
          },
```

```
            "osProfile": {
                  "adminPassword": null,
                  "adminUsername": "simple",
                  "computerName": "simple",
                  "customData": null,
                  "linuxConfiguration": {
                        "disablePasswordAuthentication": false,
                        "ssh": null
                  },
                  "secrets": [],
                  "windowsConfiguration": null
            },
            "plan": null,
            "provisioningState": "Creating",
            "resourceGroup": "VM",
            "resources": null,
            "storageProfile": {
                  "dataDisks": [],
                  "imageReference": {
                        "id": null,
                        "offer": "UbuntuServer",
                        "publisher": "Canonical",
                        "sku": "17.10",
                        "version": "latest"
                  },
                  "osDisk": {
                        "caching": "ReadWrite",
                        "createOption": "FromImage",
                        "diskSizeGb": 30,
                        "encryptionSettings": null,
                        "image": null,
                        "managedDisk": {
                              "id": "/subscriptions/9103844d-1370-4716-b02b-
69ce936865c6/resourceGroups/VM/providers/Microsoft.Compute/disks/simple_Os-
Disk_1_4da948f5ef1a4232ad2f632077326d0a",
                              "resourceGroup": "VM",
                              "storageAccountType": "Premium_LRS"
                        },
                        "name": "simple_OsDisk_1_4da948f5ef1a4232ad2f-
632077326d0a",
                        "osType": "Linux",
                        "vhd": null,
                        "writeAcceleratorEnabled": null
                  }
            },
            "tags": null,
            "type": "Microsoft.Compute/virtualMachines",
            "vmId": "6aed2e80-64b2-401b-a8a0-b82ac8a6ed5c",
            "zones": null
      },
      {
```

```
            ...
    }
]
```

Using the `--query` argument, we can specify project-specific fields to make the JSON object more useful and easier to read. This is useful if you are deserializing the JSON object into a specific type in your code:

```
az vm list --query '[].{name:name, image:storageProfile.imageReference.
offer}'
```

```
[
    {
            "image": "UbuntuServer",
            "name": "linuxvm"
    },
    {
            "image": "WindowsServer",
            "name": "winvm"
    }
]
```

Using the `[ ]` operator, you can create queries that filter your result set by comparing the values of various JSON properties:

```
az vm list --query "[?starts_with(storageProfile.imageReference.offer, 'Win-
dowsServer')]"
```

You can even combine filtering and projection to create custom queries that only return the resources you need and project only the fields that are useful to your application:

```
az vm list --query "[?starts_with(storageProfile.imageReference.offer, 'Ubun-
tu')].{name:name, id:vmId}"
```

```
[
    {
            "name": "linuxvm",
            "id": "6aed2e80-64b2-401b-a8a0-b82ac8a6ed5c"
    }
]
```

Querying resources using the fluent Azure SDK

In a manner similar to how you use the Azure CLI, you can use the Azure SDK to query resources in your subscription. The SDK may be a better option if you intend to write code to find connection information for a specific application instance. For example, you may need to write code to get the IP address of a specific VM in your subscription.

Connecting using the fluent Azure SDK

To use the APIs in the Azure management libraries for Microsoft .NET, as the first step, you need to create an authenticated client. The Azure SDK requires that you invoke the `Azure.Authenticate` static method to return an object that can fluently query resources and access their metadata. The `Authenticate` method requires a parameter that specifies an authorization file:

```
Azure azure = Azure.Authenticate("azure.auth").WithDefaultSubscription();
```

The authentication file, referenced as `azure.auth` above, contains information necessary to access your subscription using a service principal. The authorization file will look similar to the format below:

```
{
    "clientId": "b52dd125-9272-4b21-9862-0be667bdf6dc",
    "clientSecret": "ebc6e170-72b2-4b6f-9de2-99410964d2d0",
    "subscriptionId": "ffa52f27-be12-4cad-b1ea-c2c241b6cceb",
    "tenantId": "72f988bf-86f1-41af-91ab-2d7cd011db47",
    "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
    "resourceManagerEndpointUrl": "https://management.azure.com/",
    "activeDirectoryGraphResourceId": "https://graph.windows.net/",
    "sqlManagementEndpointUrl": "https://management.core.windows.
net:8443/",
    "galleryEndpointUrl": "https://gallery.azure.com/",
    "managementEndpointUrl": "https://management.core.windows.net/"
}
```

If you do not already have a service principal, you can generate a service principal and this file using the Azure CLI:

```
az ad sp create-for-rbac --sdk-auth > azure.auth
```

Listing virtual machines using the fluent Azure SDK

Once you have a variable of type `IAzure`, you can access various resources by using properties of the `IAzure` interface. For example, you can access VMs using the `VirtualMachines` property in the manner displayed below:

```
azure.VirtualMachines
```

The properties have both synchronous and asynchronous versions of methods to perform actions such as `Create`, `Delete`, `List`, and `Get`. If we wanted to get a list of VMs asynchronously, we could use the `ListAsync` method:

```
var vms = await azure.VirtualMachines.ListAsync();

foreach(var vm in vms)
{
    Console.WriteLine(vm.Name);
}
```

You can also use any language-integrated query mechanism, like language-integrated query (LINQ) in C#, to filter your VM list to a specific subset of VMs that match a filter criteria:

```
var allvms = await azure.VirtualMachines.ListAsync();

IVirtualMachine targetvm = allvms.Where(vm => vm.Name == "simple").Single-
OrDefault();

Console.WriteLine(targetvm?.Id);
```

Gathering virtual machine metadata to determine the IP address

Now that we can filter to a specific VM, we can access various properties of the `IVirtualMachine` interface and other related interfaces to get that resource's IP address.

To start, the `IVirtualMachine.GetPrimaryNetworkInterface` method implementation will return the network adapter that we need to access the VM:

```
INetworkInterface targetnic = targetvm.GetPrimaryNetworkInterface();
```

The `INetworkInterface` interface has a property named `PrimaryIPConfiguration` that will get the configuration of the primary IP address for the current network adapter:

```
INicIPConfiguration targetipconfig = targetnic.PrimaryIPConfiguration;
```

The `INicIPConfiguration` interface has a method named `GetPublicIPAddress` that will get the IP address resource that is public and associated with the current specified configuration:

```
IPublicIPAddress targetipaddress = targetipconfig.GetPublicIPAddress();
```

Finally, the `IPublicIPAddress` interface has a property named `IPAddress` that contains the current IP address as a string value:

```
Console.WriteLine($"IP Address:\t{targetipaddress.IPAddress}");
```

Your application can now use this specific IP address to communicate directly with the intended compute instance.

Implement code that handles transient faults

Transient errors

An application that communicates with elements running in the cloud has to be sensitive to the transient faults that can occur in this environment. Faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay, it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay, it might succeed.

Handling transient errors

In the cloud, transient faults aren't uncommon, and an application should be designed to handle them elegantly and transparently. This minimizes the effects faults can have on the business tasks the application is performing.

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

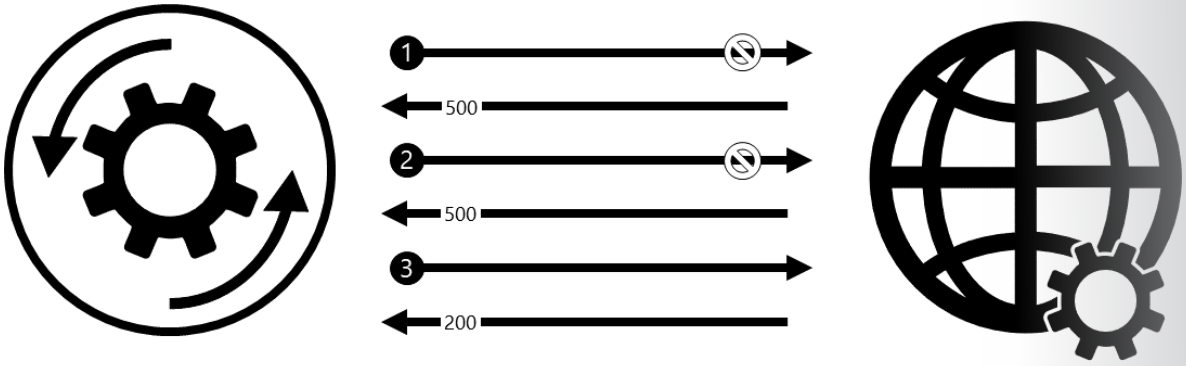
- **Cancel**: If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
- **Retry**: If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances, such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately, because the same failure is unlikely to be repeated, and the request will probably be successful.
- **Retry after a delay**: If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period of time while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable amount of time before retrying the request.

For the more common transient failures, the period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible. This reduces the chance of a busy service continuing to be overloaded. If many instances of an application are continually overwhelming a service with retry requests, it'll take the service longer to recover.

If the request still fails, the application can wait and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts, until some maximum number of requests have been attempted. The delay can be increased incrementally or exponentially depending on the type of failure and the probability that it'll be corrected during this time.

Retrying after a transient error

The following diagram illustrates invoking an operation in a hosted service using this pattern. If the request is unsuccessful after a predefined number of attempts, the application should treat the fault as an exception and handle it accordingly.



1. The application invokes an operation on a hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2. The application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3. The application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies. Some vendors provide libraries that implement retry policies, where the application can specify the maximum number of retries, the amount of time between retry attempts, and other parameters.

An application should log the details of faults and failing operations. This information is useful to operators. If a service is frequently unavailable or busy, it's often because the service has exhausted its resources. You can reduce the frequency of these faults by scaling out the service. For example, if a database service is continually overloaded, it might be beneficial to partition the database and spread the load across multiple servers.

Handling transient errors in code

This example in C# illustrates an implementation of this pattern. The `OperationWithBasicRetryAsync` method, shown below, invokes an external service asynchronously through the `TransientOperationAsync` method. The details of the `TransientOperationAsync` method will be specific to the service and are omitted from the sample code:

```
private int retryCount = 3;
private readonly TimeSpan delay = TimeSpan.FromSeconds(5);

public async Task OperationWithBasicRetryAsync()
{
    int currentRetry = 0;
    for (;;)
    {
          try
          {
                await TransientOperationAsync();
                break;
          }
          catch (Exception ex)
```

```
            {
                  Trace.TraceError("Operation Exception");
                  currentRetry++;
                  if (currentRetry > this.retryCount || !IsTransient(ex))
                  {
                        throw;
                  }
            }
            await Task.Delay(delay);
      }
}

private async Task TransientOperationAsync()
{
      ...
}
```

The statement that invokes this method is contained in a try/catch block wrapped in a for loop. The for loop exits if the call to the `TransientOperationAsync` method succeeds without throwing an exception. If the `TransientOperationAsync` method fails, the catch block examines the reason for the failure. If it's believed to be a transient error, the code waits for a short delay before retrying the operation.

The for loop also tracks the number of times that the operation has been attempted, and if the code fails three times, the exception is assumed to be more long lasting. If the exception isn't transient or it's long lasting, the catch handler will throw an exception. This exception exists in the for loop and should be caught by the code that invokes the `OperationWithBasicRetryAsync` method.

Detecting if an error is transient in code

The `IsTransient` method, shown below, checks for a specific set of exceptions that are relevant to the environment the code is run in. The definition of a transient exception will vary according to the resources being accessed and the environment the operation is being performed in:

```
private bool IsTransient(Exception ex)
{
      if (ex is OperationTransientException)
            return true;

      var webException = ex as WebException;
      if (webException != null)
      {
            return new[] {
                  WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled
            }.Contains(webException.Status);
      }

      return false;
}
```

Review questions

Module 2 review questions

Autoscale thresholds

When using autoscale why is it important to ensure the maximum and minimum values are different and have an adequate margin between them? Look at the example below, why wouldn't it be recommended to use autoscale settings with the same or very similar threshold values for out and in conditions?

- Increase instances by 1 count when Thread Count <= 600
- Decrease instances by 1 count when Thread Count >= 600

> Click to see suggested answer

Let's look at an example of what can lead to a behavior that may seem confusing. Consider the following sequence.

1. Assume there are two instances to begin with and then the average number of threads per instance grows to 625.
2. Autoscale scales out adding a third instance.
3. Next, assume that the average thread count across instances falls to 575.
4. Before scaling down, autoscale tries to estimate what the final state will be if it scaled in. For example, 575 x 3 (current instance count) = 1,725 / 2 (final number of instances when scaled down) = 862.5 threads. This means autoscale would have to immediately scale-out again even after it scaled in, if the average thread count remains the same or even falls only a small amount. However, if it scaled up again, the whole process would repeat, leading to an infinite loop.
5. To avoid this situation (termed "flapping"), autoscale does not scale down at all. Instead, it skips and reevaluates the condition again the next time the service's job executes. This can confuse many people because autoscale wouldn't appear to work when the average thread count was 575.

Estimation during a scale-in is intended to avoid "flapping" situations, where scale-in and scale-out actions continually go back and forth. Keep this behavior in mind when you choose the same thresholds for scale-out and in.

Transient errors

Why can transient errors be difficult to diagnose and fix?

> Click to see suggested answer

Transient faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay, it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay, it might succeed.

Handling transient errors

In the cloud, transient faults aren't uncommon, and an application should be designed to handle them elegantly and transparently. This minimizes the effects faults can have on the business tasks the application is performing. What are the three strategies for handling those failures?

> Click to see suggested answer

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

- **Cancel:** If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
- **Retry:** If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances, such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately, because the same failure is unlikely to be repeated, and the request will probably be successful.
- **Retry after a delay:** If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period of time while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable amount of time before retrying the request.

Module 3 Instrument solutions to support monitoring and logging

Configure instrumentation in an app or service by using Application Insights

Application Insights for web pages

Find out about the performance and usage of your web page or app. If you add Application Insights to your page script, you get timings of page loads and AJAX calls, counts and details of browser exceptions and AJAX failures, as well as users and session counts. All these can be segmented by page, client OS and browser version, geo location, and other dimensions. You can set alerts on failure counts or slow page loading. And by inserting trace calls in your JavaScript code, you can track how the different features of your web page application are used.

Add the SDK script to your app or web pages

Application Insights can be used with any web pages - you just add a short piece of JavaScript. If your web service is Java or ASP.NET, you can integrate telemetry from your server and clients.

```
<!--
To collect user behavior analytics about your application,
insert the following script into each page you want to track.
Place this code immediately before the closing </head> tag,
and before any other scripts. Your first data will appear
automatically in just a few seconds.
-->
<script type="text/javascript">
var appInsights=window.appInsights||function(a){
  function b(a){c[a]=function(){var b=arguments;c.queue.push(function()
{c[a].apply(c,b)})}}var c={config:a},d=document,e=window;setTimeout(func-
tion(){var b=d.createElement("script");b.src=a.url||"https://az416426.vo.
msecnd.net/scripts/a/ai.0.js",d.getElementsByTagName("script")[0].parent-
```

```
Node.appendChild(b)});try{c.cookie=d.cookie}catch(a){}c.queue=[];for(var
f=["Event","Exception","Metric","PageView","Trace","Dependency"];f.length;)
b("track"+f.pop());if(b("setAuthenticatedUserContext"),b("clearAuthenticat-
edUserContext"),b("startTrackEvent"),b("stopTrackEvent"),b("startTrack-
Page"),b("stopTrackPage"),b("flush"),!a.disableExceptionTracking){f="oner-
ror",b("_"+f);var g=e[f];e[f]=function(a,b,d,e,h){var
i=g&&g(a,b,d,e,h);return!0!==i&&c["_"+f](a,b,d,e,h),i}}return c
  }({
      instrumentationKey:"<your instrumentation key>"
  });

window.appInsights=appInsights,appInsights.queue&&0===appInsights.queue.
length&&appInsights.trackPageView();
</script>
```

Insert the script just before the `</head>` tag of every page you want to track. If your website has a master page, you can put the script there. For example in an ASP.NET MVC project, you'd put it in `View\Shared\_Layout.cshtml`.

The script contains the instrumentation key that directs the data to your Application Insights resource.

Detailed configuration

There are several parameters you can set, though in most cases, you shouldn't need to. For example, you can disable or limit the number of Ajax calls reported per page view (to reduce traffic). Or you can set debug mode to have telemetry move rapidly through the pipeline without being batched.

To set these parameters, look for this line in the code snippet, and add more comma-separated items after it:

```
})({
  instrumentationKey: "..."
  // Insert here
});
```

The **available parameters**[1] include:

```
// Send telemetry immediately without batching.
// Remember to remove this when no longer required, as it
// can affect browser performance.
enableDebug: boolean,

// Don't log browser exceptions.
disableExceptionTracking: boolean,

// Don't log ajax calls.
disableAjaxTracking: boolean,

// Limit number of Ajax calls logged, to reduce traffic.
maxAjaxCallsPerView: 10, // default is 500
```

1 https://github.com/Microsoft/ApplicationInsights-JS/blob/master/API-reference.md#config

```
// Time page load up to execution of first trackPageView().
overridePageViewDuration: boolean,

// Set dynamically for an authenticated user.
accountId: string,
```

Application Insights for .NET console applications

To get started:

- In the Azure portal, create an Application Insights resource. For application type, choose **General**.
- Take a copy of the Instrumentation Key. Find the key in the **Essentials** drop-down of the new resource you created.
- Install latest Microsoft.ApplicationInsights package.
- Set the instrumentation key in your code before tracking any telemetry (or set `APPINSIGHTS_INSTRUMENTATIONKEY` environment variable). After that, you should be able to manually track telemetry and see it on the Azure portal

```
TelemetryConfiguration.Active.InstrumentationKey = " *your key* ";
var telemetryClient = new TelemetryClient();
telemetryClient.TrackTrace("Hello World!");
```

- Install latest version of **Microsoft.ApplicationInsights.DependencyCollector**[2] package - it automatically tracks HTTP, SQL, or some other external dependency calls.

You may initialize and configure Application Insights from the code or using ApplicationInsights.config file. Make sure initialization happens as early as possible.

Note: Instructions referring to ApplicationInsights.config are only applicable to apps that are targeting the .NET Framework, and do not apply to .NET Core applications.

Using config file

By default, Application Insights SDK looks for ApplicationInsights.config file in the working directory when TelemetryConfiguration is being created

```
// Reads ApplicationInsights.config file if present
TelemetryConfiguration config = TelemetryConfiguration.Active;
```

You may also specify path to the config file.

```
using System.IO;
TelemetryConfiguration configuration = TelemetryConfiguration.CreateFromConfig-
uration(File.ReadAllText("C:\\ApplicationInsights.config"));
var telemetryClient = new TelemetryClient(configuration);
```

For more information, see **configuration file reference**[3].

2 https://www.nuget.org/packages/Microsoft.ApplicationInsights.DependencyCollector
3 https://docs.microsoft.com/en-us/azure/azure-monitor/app/configuration-with-applicationinsights-config

You may get a full example of the config file by installing latest version of **Microsoft.ApplicationInsights.WindowsServer**[4] package. Here is the minimal configuration for dependency collection that is equivalent to the code example.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationInsights xmlns="http://schemas.microsoft.com/ApplicationIn-
sights/2013/Settings">
  <InstrumentationKey>Your Key</InstrumentationKey>
  <TelemetryInitializers>
    <Add Type="Microsoft.ApplicationInsights.DependencyCollector.HttpDe-
pendenciesParsingTelemetryInitializer, Microsoft.AI.DependencyCollector"/>
  </TelemetryInitializers>
  <TelemetryModules>
    <Add Type="Microsoft.ApplicationInsights.DependencyCollector.Dependen-
cyTrackingTelemetryModule, Microsoft.AI.DependencyCollector">
      <ExcludeComponentCorrelationHttpHeadersOnDomains>
        <Add>core.windows.net</Add>
        <Add>core.chinacloudapi.cn</Add>
        <Add>core.cloudapi.de</Add>
        <Add>core.usgovcloudapi.net</Add>
        <Add>localhost</Add>
        <Add>127.0.0.1</Add>
      </ExcludeComponentCorrelationHttpHeadersOnDomains>
      <IncludeDiagnosticSourceActivities>
        <Add>Microsoft.Azure.ServiceBus</Add>
        <Add>Microsoft.Azure.EventHubs</Add>
      </IncludeDiagnosticSourceActivities>
    </Add>
  </TelemetryModules>
  <TelemetryChannel Type="Microsoft.ApplicationInsights.WindowsServer.
TelemetryChannel.ServerTelemetryChannel, Microsoft.AI.ServerTelemetryChan-
nel"/>
</ApplicationInsights>
```

Configuring telemetry collection from code

- During application start-up create and configure `DependencyTrackingTelemetryModule` instance - it must be singleton and must be preserved for application lifetime.

```
var module = new DependencyTrackingTelemetryModule();

// Prevent Correlation Id to be sent to certain endpoints.
// You may add other domains as needed.
module.ExcludeComponentCorrelationHttpHeadersOnDomains.Add("core.windows.
net");
//...

// enable known dependency tracking, note that in future versions, we will
extend this list.
```

4 https://www.nuget.org/packages/Microsoft.ApplicationInsights.WindowsServer

```
// please check default settings in
// https://github.com/Microsoft/ApplicationInsights-dotnet-server/blob/
develop/Src/DependencyCollector/NuGet/ApplicationInsights.config.install.
xdt#L20
module.IncludeDiagnosticSourceActivities.Add("Microsoft.Azure.ServiceBus");
module.IncludeDiagnosticSourceActivities.Add("Microsoft.Azure.EventHubs");
//....

// initialize the module
module.Initialize(configuration);
```

- Add common telemetry initializers

```
// stamps telemetry with correlation identifiers
TelemetryConfiguration.Active.TelemetryInitializers.Add(new OperationCorre-
lationTelemetryInitializer());

// ensures proper DependencyTelemetry.Type is set for Azure RESTful API
calls
TelemetryConfiguration.Active.TelemetryInitializers.Add(new HttpDependen-
ciesParsingTelemetryInitializer());
```

- For .NET Framework Windows app, you may also install and initialize Performance Counter collector module.

Full example

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DependencyCollector;
using Microsoft.ApplicationInsights.Extensibility;
using System.Net.Http;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            TelemetryConfiguration configuration = TelemetryConfiguration.
Active;

            configuration.InstrumentationKey = "removed";
            configuration.TelemetryInitializers.Add(new OperationCorrela-
tionTelemetryInitializer());
            configuration.TelemetryInitializers.Add(new HttpDependen-
ciesParsingTelemetryInitializer());

            var telemetryClient = new TelemetryClient();
            using (InitializeDependencyTracking(configuration))
            {
```

```
                // run app...

                telemetryClient.TrackTrace("Hello World!");

                using (var httpClient = new HttpClient())
                {
                    // Http dependency is automatically tracked!
                    httpClient.GetAsync("https://microsoft.com").Wait();
                }

            }

            // before exit, flush the remaining data
            telemetryClient.Flush();

            // flush is not blocking so wait a bit
            Task.Delay(5000).Wait();

        }

        static DependencyTrackingTelemetryModule InitializeDependencyTrack-
ing(TelemetryConfiguration configuration)
        {
            var module = new DependencyTrackingTelemetryModule();

            // prevent Correlation Id to be sent to certain endpoints. You
may add other domains as needed.
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.Ad-
d("core.windows.net");
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.Ad-
d("core.chinacloudapi.cn");
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.Ad-
d("core.cloudapi.de");
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.Ad-
d("core.usgovcloudapi.net");
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.Add("lo-
calhost");
            module.ExcludeComponentCorrelationHttpHeadersOnDomains.
Add("127.0.0.1");

            // enable known dependency tracking, note that in future ver-
sions, we will extend this list.
            // please check default settings in https://github.com/Micro-
soft/ApplicationInsights-dotnet-server/blob/develop/Src/DependencyCollec-
tor/NuGet/ApplicationInsights.config.install.xdt#L20
            module.IncludeDiagnosticSourceActivities.Add("Microsoft.Azure.
ServiceBus");
            module.IncludeDiagnosticSourceActivities.Add("Microsoft.Azure.
EventHubs");

            // initialize the module
```

```
            module.Initialize(configuration);

            return module;
        }
    }
}
```

Application Insights in classic Windows desktop apps

Applications hosted on premises, in Azure, and in other clouds can all take advantage of Application Insights. The only limitation is the need to allow communication to the Application Insights service.

To send telemetry to Application Insights from a Classic Windows application

1. In the Azure portal, create an Application Insights resource. For application type, choose ASP.NET app.
2. Take a copy of the Instrumentation Key. Find the key in the Essentials drop-down of the new resource you just created.
3. In Visual Studio, edit the NuGet packages of your app project, and add Microsoft.ApplicationInsights.WindowsServer. (Or choose Microsoft.ApplicationInsights if you just want the bare API, without the standard telemetry collection modules.)
4. Set the instrumentation key either in your code:
5. `TelemetryConfiguration.Active.InstrumentationKey = " your key ";`
6. ***or*** in ApplicationInsights.config (if you installed one of the standard telemetry packages):
7. `<InstrumentationKey>`*your key*`</InstrumentationKey>`
8. If you use ApplicationInsights.config, make sure its properties in Solution Explorer are set to **Build Action = Content**, **Copy to Output Directory = Copy**.
9. Use the API to send telemetry.
10. Run your app, and see the telemetry in the resource you created in the Azure Portal.

Example code

```
public partial class Form1 : Form
{
    private TelemetryClient tc = new TelemetryClient();
    ...
    private void Form1_Load(object sender, EventArgs e)
    {
        // Alternative to setting ikey in config file:
        tc.InstrumentationKey = "key copied from portal";

        // Set session data:
        tc.Context.User.Id = Environment.UserName;
        tc.Context.Session.Id = Guid.NewGuid().ToString();
```

```
        tc.Context.Device.OperatingSystem = Environment.OSVersion.To-
String();

        // Log a page view:
        tc.TrackPageView("Form1");
        ...
    }

    protected override void OnClosing(CancelEventArgs e)
    {
        stop = true;
        if (tc != null)
        {
            tc.Flush(); // only for desktop apps

            // Allow time for flushing:
            System.Threading.Thread.Sleep(1000);
        }
        base.OnClosing(e);
    }
```

Analyze and troubleshoot solutions by using Azure Monitor

Application Map: Triage Distributed Applications

Application Map helps you spot performance bottlenecks or failure hotspots across all components of your distributed application. Each node on the map represents an application component or its dependencies; and has health KPI and alerts status. You can click through from any component to more detailed diagnostics, such as Application Insights events. If your app uses Azure services, you can also click through to Azure diagnostics, such as SQL Database Advisor recommendations.

What is a Component?

Components are independently deployable parts of your distributed/microservices application. Developers and operations teams have code-level visibility or access to telemetry generated by these application components.

- Components are different from "observed" external dependencies such as SQL, EventHub etc. which your team/organization may not have access to (code or telemetry).
- Components run on any number of server/role/container instances.
- Components can be separate Application Insights instrumentation keys (even if subscriptions are different) or different roles reporting to a single Application Insights instrumentation key. The preview map experience shows the components regardless of how they are set up.

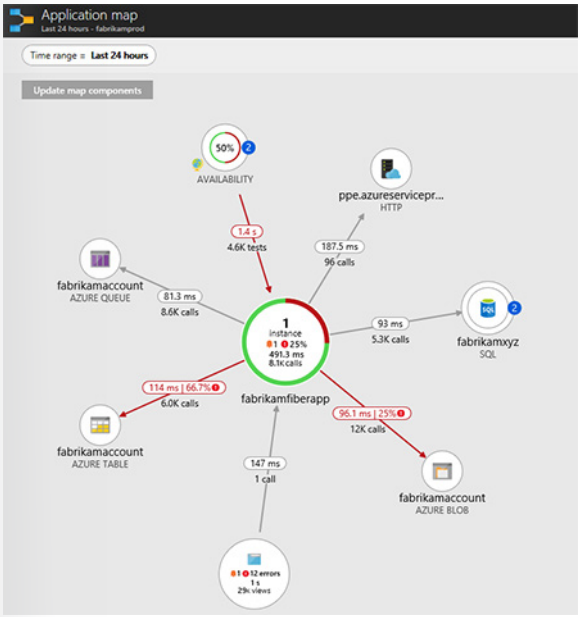
Composite Application Map

You can see the full application topology across multiple levels of related application components. Components could be different Application Insights resources, or different roles in a single resource. The app map finds components by following HTTP dependency calls made between servers with the Application Insights SDK installed.

This experience starts with progressive discovery of the components. When you first load the application map, a set of queries are triggered to discover the components related to this component. A button at the top-left corner will update with the number of components in your application as they are discovered.

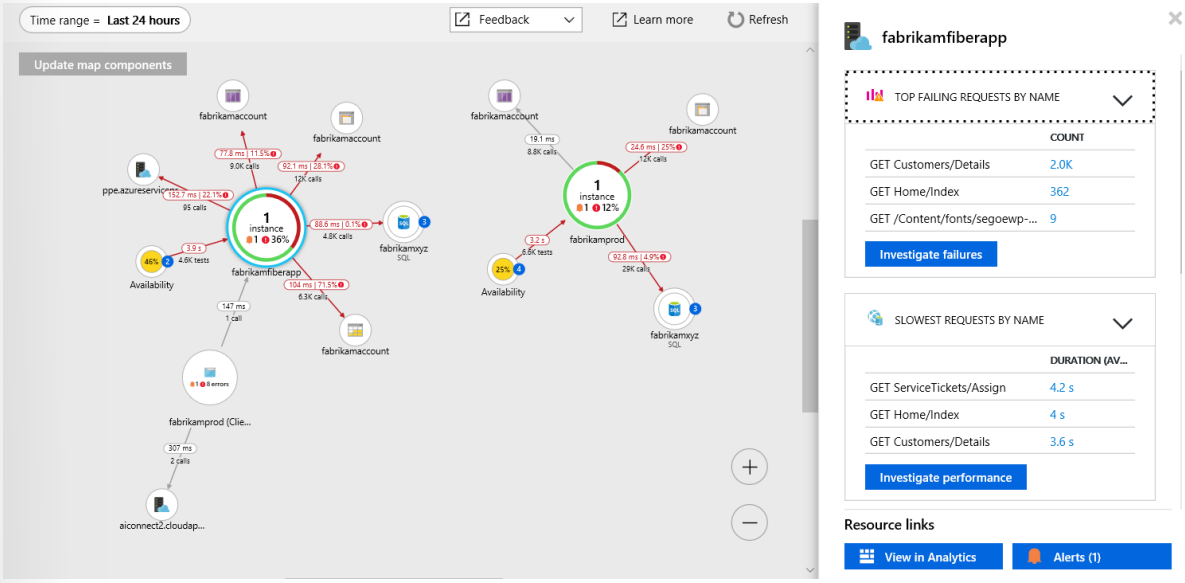
On clicking **Update map components**, the map is refreshed with all components discovered until that point. Depending on the complexity of your application, this may take a minute to load.

If all of the components are roles within a single Application Insights resource, then this discovery step is not required. The initial load for such an application will have all its components.



One of the key objectives with this experience is to be able to visualize complex topologies with hundreds of components.

Click on any component to see related insights and go to the performance and failure triage experience for that component.



Set cloud_RoleName

Application Map uses the `cloud_RoleName` property to identify the components on the map. The Application Insights SDK automatically adds the `cloud_RoleName` property to the telemetry emitted by components. For example, the SDK will add a web site name or service role name to the `cloud_RoleName` property. However, there are cases where you may want to override the default value. To override `cloud_RoleName` and change what gets displayed on the Application Map:

.NET

```
using Microsoft.ApplicationInsights.Channel;
using Microsoft.ApplicationInsights.Extensibility;

namespace CustomInitializer.Telemetry
{
    public class MyTelemetryInitializer : ITelemetryInitializer
    {
        public void Initialize(ITelemetry telemetry)
        {
            if (string.IsNullOrEmpty(telemetry.Context.Cloud.RoleName))
            {
                //set custom role name here
                telemetry.Context.Cloud.RoleName = "RoleName";
            }
        }
    }
}
```

Load your initializer

In *ApplicationInsights.config*:

```
    <ApplicationInsights>
      <TelemetryInitializers>
        <!-- Fully qualified type name, assembly name: -->
        <Add Type="CustomInitializer.Telemetry.MyTelemetryInitializer,
CustomInitializer"/>
        ...
      </TelemetryInitializers>
    </ApplicationInsights>
```

An alternate method is to instantiate the initializer in code, for example in Global.aspx.cs:

```
 using Microsoft.ApplicationInsights.Extensibility;
 using CustomInitializer.Telemetry;

    protected void Application_Start()
    {
        // ...
        TelemetryConfiguration.Active.TelemetryInitializers.Add(new MyTelem-
etryInitializer());
    }
```

Node.js

```
var appInsights = require("applicationinsights");
appInsights.setup('INSTRUMENTATION_KEY').start();
appInsights.defaultClient.context.tags["ai.cloud.role"] = "your role name";
```

```
appInsights.defaultClient.context.tags["ai.cloud.roleInstance"] = "your
role instance";
```

Alternate method for Node.js

```
var appInsights = require("applicationinsights");
appInsights.setup('INSTRUMENTATION_KEY').start();

appInsights.defaultClient.addTelemetryProcessor(envelope => {
    envelope.tags["ai.cloud.role"] = "your role name";
    envelope.tags["ai.cloud.roleInstance"] = "your role instance"
});
```

Client/browser-side JavaScript

```
appInsights.queue.push(() => {
appInsights.context.addTelemetryInitializer((envelope) => {
  envelope.tags["ai.cloud.role"] = "your role name";
  envelope.tags["ai.cloud.roleInstance"] = "your role instance";
});
});
```

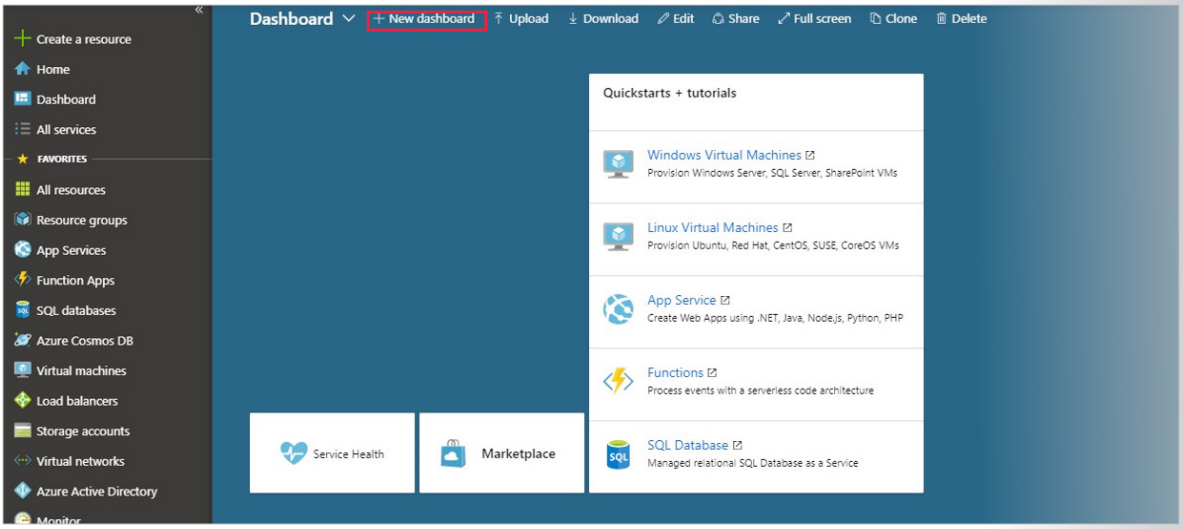
Create custom KPI dashboards

You can create multiple dashboards in the Azure portal that each include tiles visualizing data from multiple Azure resources across different resource groups and subscriptions. You can pin different charts and views from Azure Application Insights to create custom dashboards that provide you with complete picture of the health and performance of your application. This lesson walks you through the creation of a custom dashboard that includes multiple types of data and visualizations from Azure Application Insights.

Create a new dashboard

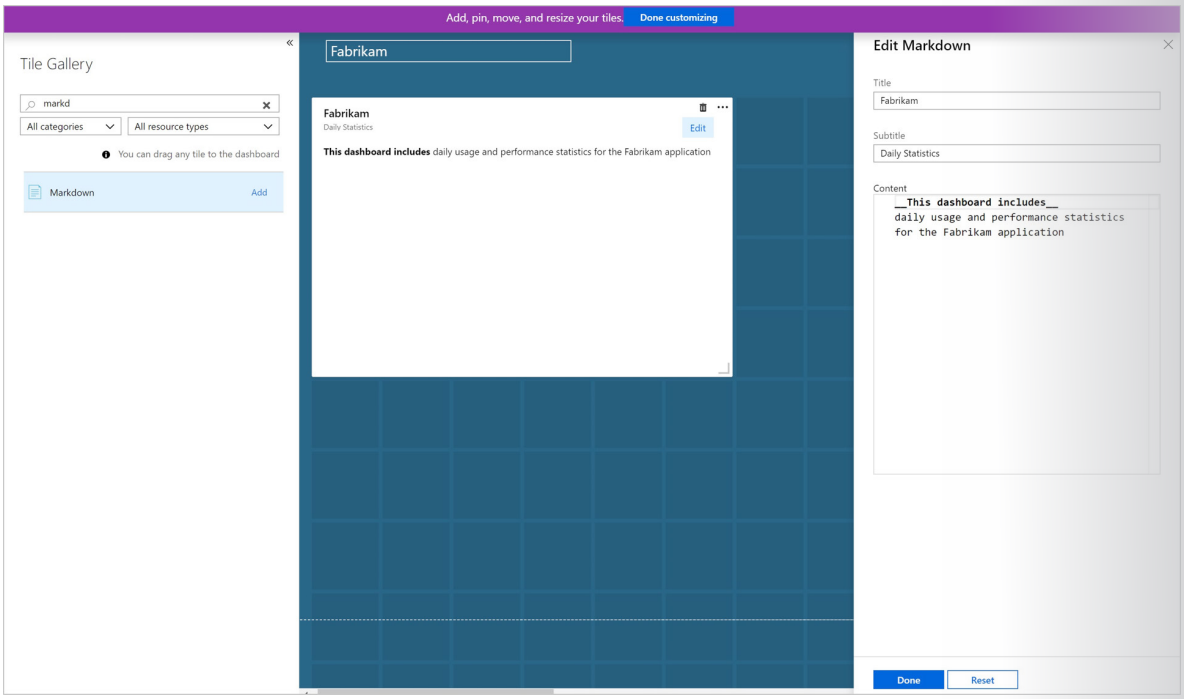
A single dashboard can contain resources from multiple applications, resource groups, and subscriptions. To create a new dashboard:

1. On the dashboard pane, select **New dashboard**.



2.

3. Type a name for the dashboard.
4. Have a look at the **Tile Gallery** for a variety of tiles that you can add to your dashboard. In addition to adding tiles from the gallery you can pin charts and other views directly from Application Insights to the dashboard.
5. Locate the **Markdown** tile and drag it on to your dashboard. This tile allows you to add text formatted in markdown which is ideal for adding descriptive text to your dashboard.
6. Add text to the tile's properties and resize it on the dashboard canvas.



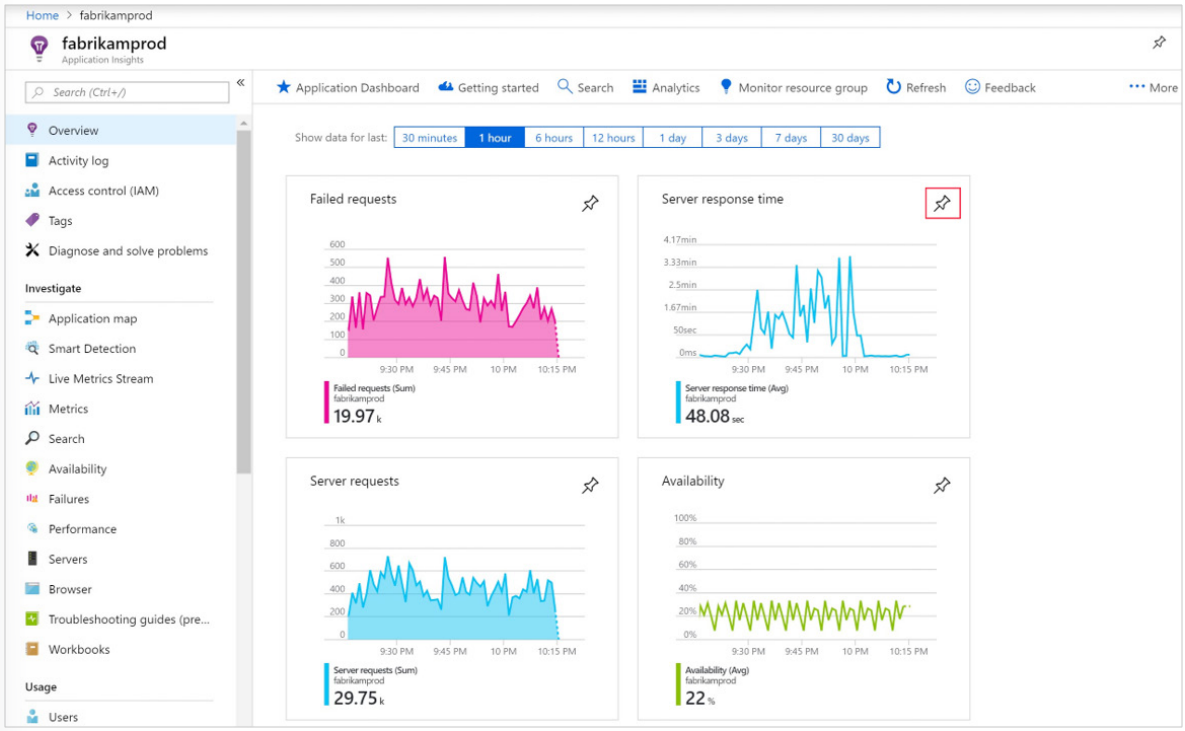
7.

8. Click **Done customizing** at the top of the screen to exit tile customization mode.

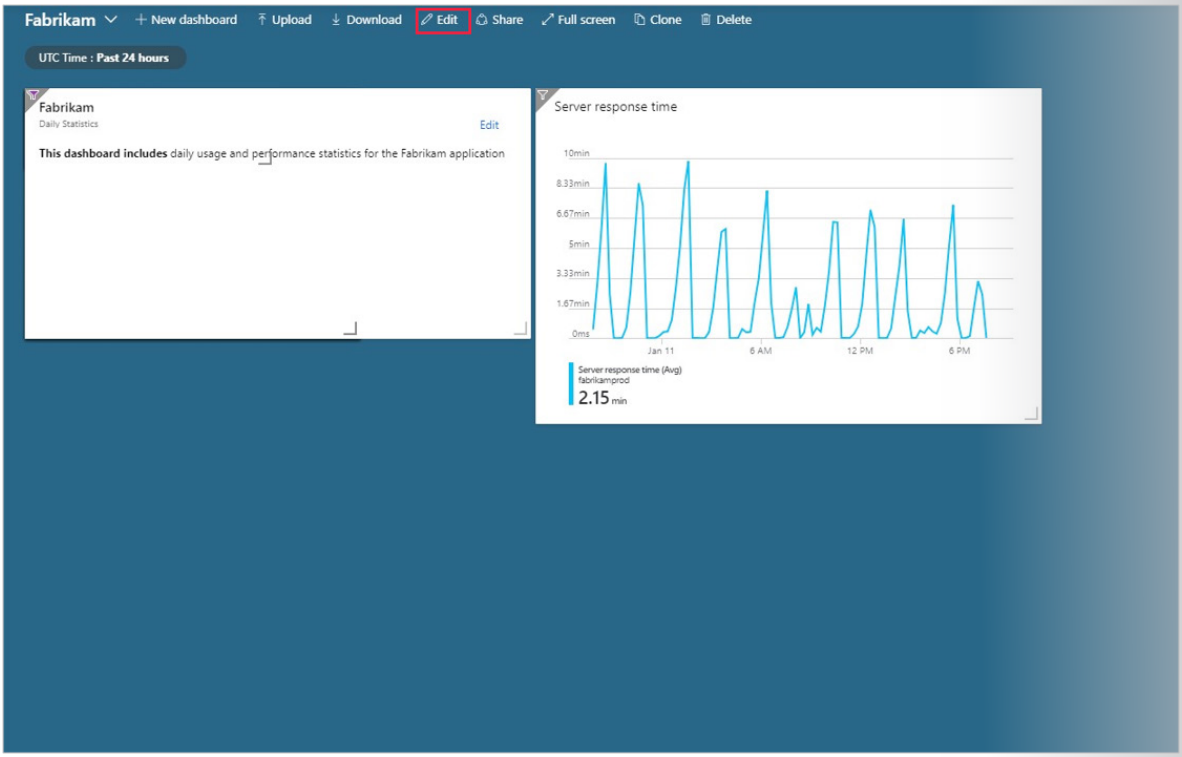
Add health overview

You can add Application Insights tiles from the Tile Gallery, or you can pin them directly from Application Insights screens. This allows you to configure charts and views that you're already familiar with before pinning them to your dashboard. Start by adding the standard health overview for your application. This requires no configuration and allows minimal customization in the dashboard.

1. Select your **Application Insights** resource on the home screen.
2. In the **Overview** pane, click the pushpin icon to add the tile to the last dashboard that you were viewing.



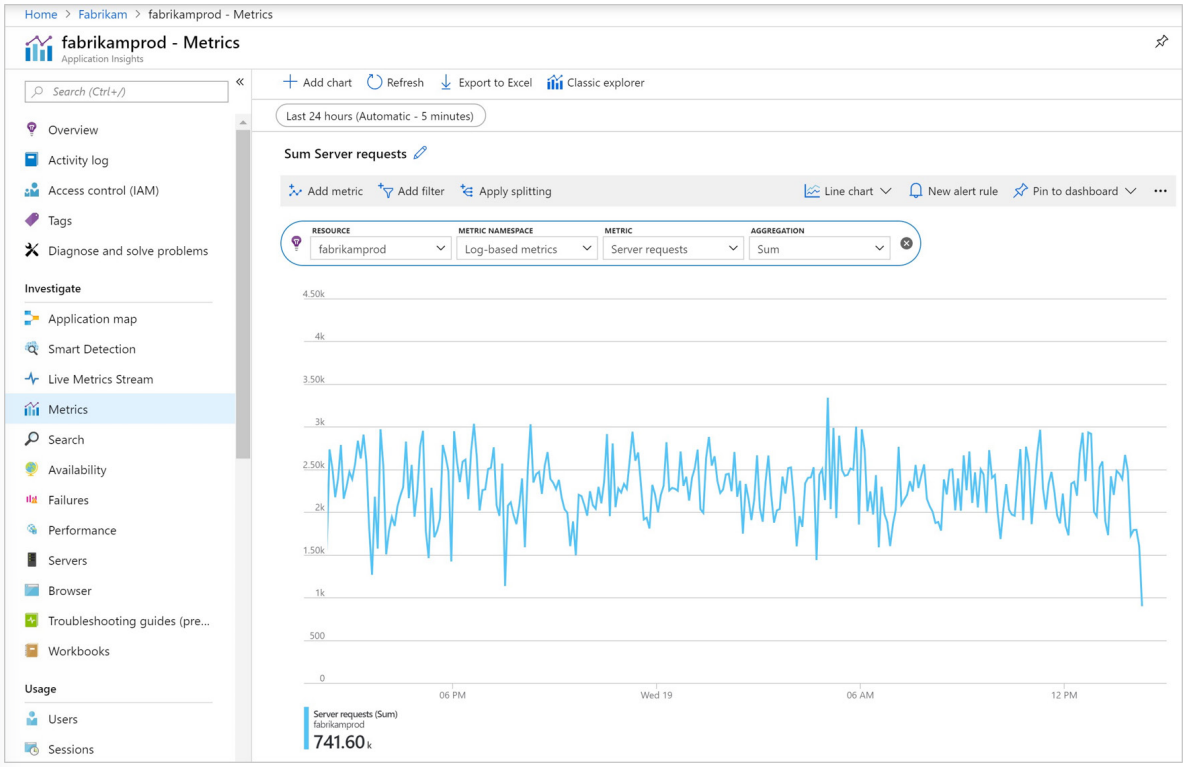
3.
4. In the top right a notification will appear that your tile was pinned to your dashboard. Click **Pinned to dashboard** in the notification to return to your dashboard or use the dashboard pane.
5. That tile is now added to your dashboard. Select **Edit** to change the positioning of the tile. Click and drag the it into position and then click **Done customizing**. Your dashboard now has a tile with some useful information.

6. 

Add custom metric chart

The **Metrics** panel allows you to graph a metric collected by Application Insights over time with optional filters and grouping. Like everything else in Application Insights, you can add this chart to the dashboard. This does require you to do a little customization first.

1. Select your **Application Insights** resource in the home screen.
2. Select **Metrics**.
3. An empty chart has already been created, and you're prompted to add a metric. Add a metric to the chart and optionally add a filter and a grouping. The example below shows the number of server requests grouped by success. This gives a running view of successful and unsuccessful requests.

4. 

5. Select P**in to dashboard** on the right. This adds the view to the last dashboard that you were working with.

6. In the top right a notification will appear that your tile was pinned to your dashboard. Click **Pinned to dashboard** in the notification to return to your dashboard or use the dashboard blade.

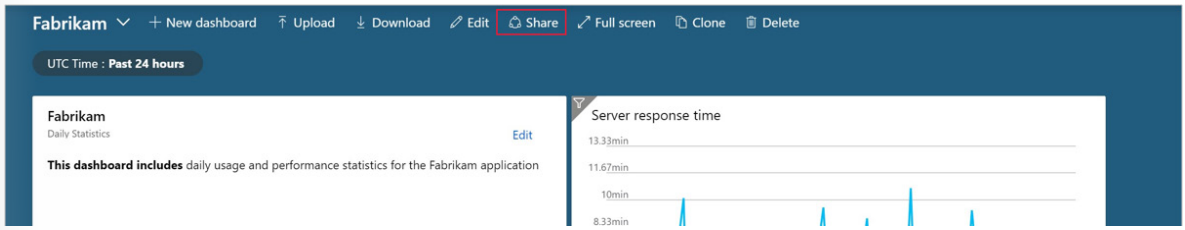
7. That tile is now added to your dashboard.

Add Analytics query

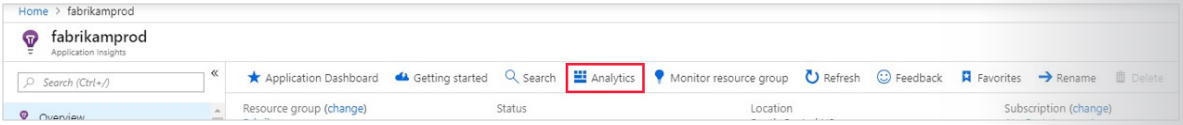
Azure Application Insights Analytics provides a rich query language that allows you to analyze all of the data collected Application Insights. Just like charts and other views, you can add the output of an Analytics query to your dashboard.

Since Azure Applications Insights Analytics is a separate service, you need to share your dashboard for it to include an Analytics query. When you share an Azure dashboard, you publish it as an Azure resource which can make it available to other users and resources.

1. At the top of the dashboard screen, click **Share**.

2. 

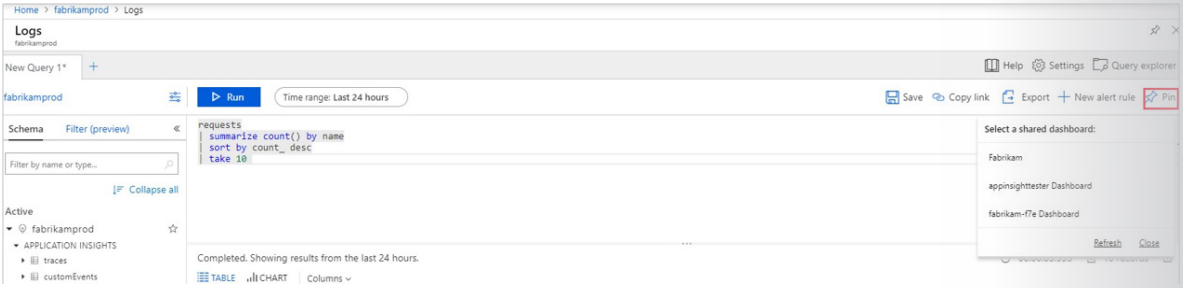
3. Keep the **Dashboard name** the same and select the **Subscription Name** to share the dashboard. Click **Publish**. The dashboard is now available to other services and subscriptions. You can optionally define specific users who should have access to the dashboard.
4. Select your **Application Insights** resource in the home screen.
5. Click **Analytics** at the top of the screen to open the Analytics portal.



6.
7. Type the following query, which returns the top 10 most requested pages and their request count:

```
requests
| summarize count() by name
| sort by count_ desc
| take 10
```

8. Click **Run** to validate the results of the query.
9. Click the pin icon and select the name of your dashboard. The reason that this option has you select a dashboard unlike the previous steps where the last dashboard was used is because the Analytics console is a separate service and needs to select from all available shared dashboards.



10.

View activity logs to audit actions on resources

Through activity logs, you can determine:

- what operations were taken on the resources in your subscription
- who initiated the operation (although operations initiated by a backend service do not return a user as the caller)
- when the operation occurred
- the status of the operation
- the values of other properties that might help you research the operation

The activity log contains all write operations (`PUT`, `POST`, `DELETE`) performed on your resources. It does not include read operations (`GET`). For a list of resource actions, see Azure Resource Manager Resource Provider operations. You can use the audit logs to find an error when troubleshooting or to monitor how a user in your organization modified a resource.

Activity logs are retained for 90 days. You can query for any range of dates, as long as the starting date is not more than 90 days in the past.

You can retrieve information from the activity logs through the portal, PowerShell, Azure CLI, Insights REST API, or Insights .NET Library.

PowerShell

1. To retrieve log entries, run the `Get-AzureRmLog` command. You provide additional parameters to filter the list of entries. If you do not specify a start and end time, entries for the last hour are returned. For example, to retrieve the operations for a resource group during the past hour run:

```
Get-AzureRmLog -ResourceGroup ExampleGroup
```

1. The following example shows how to use the activity log to research operations taken during a specified time. The start and end dates are specified in a date format.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime 2015-08-28T06:00
-EndTime 2015-09-10T06:00
```

1. Or, you can use date functions to specify the date range, such as the last 14 days.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime (Get-Date).Ad-
dDays(-14)
```

2. Depending on the start time you specify, the previous commands can return a long list of operations for the resource group. You can filter the results for what you are looking for by providing search criteria. For example, if you are trying to research how a web app was stopped, you could run the following command:

```
Get-AzureRmLog -ResourceGroup ExampleGroup -StartTime (Get-Date).Ad-
dDays(-14) | Where-Object OperationName -eq Microsoft.Web/sites/stop/action
```

3. Which for this example shows that a stop action was performed by someone@contoso.com.

```
Authorization     :
Scope     : /subscriptions/xxxxx/resourcegroups/ExampleGroup/providers/
Microsoft.Web/sites/ExampleSite
Action    : Microsoft.Web/sites/stop/action
Role      : Subscription Admin
Condition :
Caller            : someone@contoso.com
CorrelationId     : 84beae59-92aa-4662-a6fc-b6fecc0ff8da
EventSource       : Administrative
EventTimestamp    : 8/28/2015 4:08:18 PM
OperationName     : Microsoft.Web/sites/stop/action
ResourceGroupName : ExampleGroup
ResourceId        : /subscriptions/xxxxx/resourcegroups/ExampleGroup/
providers/Microsoft.Web/sites/ExampleSite
Status            : Succeeded
SubscriptionId    : xxxxx
SubStatus         : OK
```

4. You can look up the actions taken by a particular user, even for a resource group that no longer exists.

```
Get-AzureRmLog -ResourceGroup deletedgroup -StartTime (Get-Date).Ad-
dDays(-14) -Caller someone@contoso.com
```

5. You can filter for failed operations.

```
Get-AzureRmLog -ResourceGroup ExampleGroup -Status Failed
```

6. You can focus on one error by looking at the status message for that entry.

```
((Get-AzureRmLog -Status Failed -ResourceGroup ExampleGroup -DetailedOut-
put).Properties[1].Content["statusMessage"] | ConvertFrom-Json).error
```

7. Which returns:

```
code           message
----           -------
DnsRecordInUse DNS record dns.westus.cloudapp.azure.com is already used by
another public IP.
```

Azure CLI

To retrieve log entries, run the az monitor activity-log list command.

```
az monitor activity-log list --resource-group <group name>
```

REST API

The REST operations for working with the activity log are part of the Insights REST API. To retrieve activity log events, see **List the management events in a subscription**[5].

Monitor availability and responsiveness of any web site

After you've deployed your web app or web site to any server, you can set up tests to monitor its availability and responsiveness. Azure Application Insights sends web requests to your application at regular intervals from points around the world. It alerts you if your application doesn't respond, or responds slowly.

You can set up availability tests for any HTTP or HTTPS endpoint that is accessible from the public internet. You don't have to add anything to the web site you're testing. It doesn't even have to be your site: you could test a REST API service on which you depend.

There are two types of availability tests:

- URL ping test: a simple test that you can create in the Azure portal.
- Multi-step web test: which you create in Visual Studio Enterprise and upload to the portal.

You can create up to 100 availability tests per application resource.

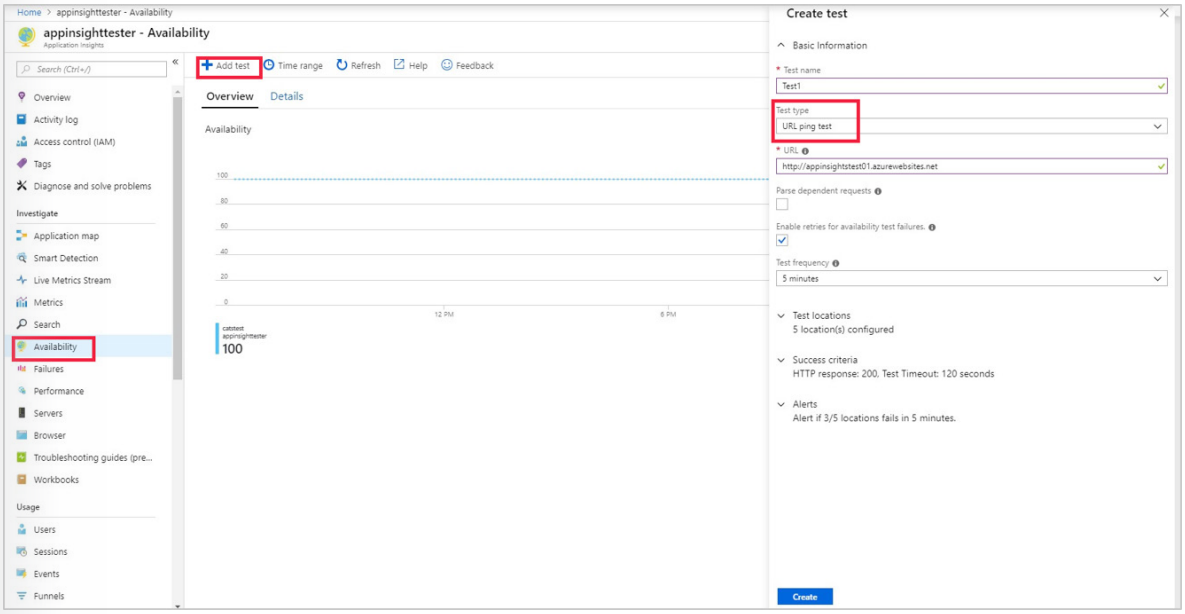
5 https://msdn.microsoft.com/library/azure/dn931934.aspx

If you have already configured Application Insights for your web app, open its Application Insights resource in the Azure portal.

Or, if you want to see your reports in a new resource, go to the Azure portal, and create an Application Insights resource.

Create a URL ping test

Open the **Availability** blade and add a test.



- **URL:** Can be any web page you want to test, but it must be visible from the public internet. The URL can include a query string. So, for example, you can exercise your database a little. If the URL resolves to a redirect, we follow it up to 10 redirects.
- **Parse dependent requests:** If this option is checked, the test requests images, scripts, style files, and other files that are part of the web page under test. The recorded response time includes the time taken to get these files. The test fails if all these resources cannot be successfully downloaded within the timeout for the whole test. If the option is not checked, the test only requests the file at the URL you specified.
- **Enable retries:** If this option is checked, when the test fails, it is retried after a short interval. A failure is reported only if three successive attempts fail. Subsequent tests are then performed at the usual test frequency. Retry is temporarily suspended until the next success. This rule is applied independently at each test location. We recommend this option. On average, about 80% of failures disappear on retry.
- **Test frequency:** Sets how often the test is run from each test location. With a default frequency of five minutes and five test locations, your site is tested on average every minute.
- **Test locations** are the places from where our servers send web requests to your URL. Our minimum number of recommended test locations is five in order to insure that you can distinguish problems in your website from network issues. You can select up to 16 locations.
- **Note:** We strongly recommend testing from multiple locations with a minimum of five locations. This is to prevent false alarms that may result from transient issues with a specific location. In addition we have found that the optimal configuration is to have the number of test locations be equal to the alert

location threshold + 2. Enabling the "Parse dependent requests" option results in a stricter check. The test could fail for cases which may not be noticeable when manually browsing the site.

- **Success criteria:**
- **Test timeout:** Decrease this value to be alerted about slow responses. The test is counted as a failure if the responses from your site have not been received within this period. If you selected Parse dependent requests, then all the images, style files, scripts, and other dependent resources must have been received within this period.
- **HTTP response:** The returned status code that is counted as a success. 200 is the code that indicates that a normal web page has been returned.
- **Content match:** a string, like "Welcome!" We test that an exact case-sensitive match occurs in every response. It must be a plain string, without wildcards. Don't forget that if your page content changes you might have to update it.
- **Alert location threshold:** We recommend a minimum of 3/5 locations. The optimal relationship between alert location threshold and the number of test locations is alert location threshold = number of test locations - 2, with a minimum of five test locations.

Multi-step web tests

You can monitor a scenario that involves a sequence of URLs. For example, if you are monitoring a sales website, you can test that adding items to the shopping cart works correctly.

To create a multi-step test, you record the scenario by using Visual Studio Enterprise, and then upload the recording to Application Insights. Application Insights replays the scenario at intervals and verifies the responses.

- You can't use coded functions or loops in your tests. The test must be contained completely in the .webtest script. However, you can use standard plugins.
- Only English characters are supported in the multi-step web tests. If you use Visual Studio in other languages, please update the web test definition file to translate/exclude non-English characters.

Review questions

Module 3 review questions

Application Insights for web pages

What types of information is gathered if you add the SDK script to your app or web page?

> Click to see suggested answer

If you add Application Insights to your page script, you get timings of page loads and AJAX calls, counts and details of browser exceptions and AJAX failures, as well as users and session counts. All these can be segmented by page, client OS and browser version, geo location, and other dimensions. You can set alerts on failure counts or slow page loading. And by inserting trace calls in your JavaScript code, you can track how the different features of your web page application are used.

Application Insights data collection

Can Application Insights be used to monitor classic Windows desktop apps in a corporate (non-cloud) environment?

> Click to see suggested answer

Applications hosted on premises, in Azure, and in other clouds can all take advantage of Application Insights. The only limitation is the need to allow communication to the Application Insights service.

Module 4 Integrate caching and content delivery within solutions

Azure Cache for Redis

Azure Cache for Redis overview

Azure Cache for Redis is based on the popular software Redis. It is typically used as a cache to improve the performance and scalability of systems that rely heavily on backend data-stores. Performance is improved by temporarily copying frequently accessed data to fast storage located close to the application. With Azure Cache for Redis, this fast storage is located in-memory with Azure Cache for Redis instead of being loaded from disk by a database.

Azure Cache for Redis can also be used as an in-memory data structure store, distributed non-relational database, and message broker. Application performance is improved by taking advantage of the low-latency, high-throughput performance of the Redis engine.

Azure Cache for Redis gives you access to a secure, dedicated Azure Cache for Redis, managed by Microsoft, hosted within Azure, and accessible to any application within or outside of Azure.

What type of data can be stored in the cache?

Redis supports a variety of data types all oriented around binary safe strings. This means that you can use any binary sequence for a value, from a string like "i-love-rocky-road" to the contents of an image file. An empty string is also a valid value.

- Binary-safe strings (most common)
- Lists of strings
- Unordered sets of strings
- Hashes
- Sorted sets of strings
- Maps of strings

Each data value is associated to a key which can be used to lookup the value from the cache. Redis works best with smaller values (100k or less), so consider chopping up bigger data into multiple keys. Storing larger values is possible (up to 500 MB), but increases network latency and can cause caching and out-of-memory issues if the cache isn't configured to expire old values.

What is a Redis key?

Redis keys are also binary safe strings. Here are some guidelines for choosing keys:

- Avoid long keys. They take up more memory and require longer lookup times because they have to be compared byte-by-byte. If you want to use a binary blob as the key, generate a unique hash and use that as the key instead. The maximum size of a key is 512 MB, but you should never use a key that size.
- Use keys which can identify the data. For example, "sport:football;date:2008-02-02" would be a better key than "fb:8-2-2". The former is more readable and the extra size is negligible. Find the balance between size and readability.
- Use a convention. A good one is "object:id", as in "sport:football".

How is data stored in a Redis cache?

Data in Redis is stored in ***nodes*** and ***clusters***.

Nodes are a space in Redis where your data is stored.

Clusters are sets of three or more nodes your dataset is split across. Clusters are useful because your operations will continue if a node fails or is unable to communicate to the rest of the cluster.

What are Redis caching architectures?

Redis caching architecture is how we distribute our data in the cache. Redis distributes data in three major ways:

1. Single node
2. Multiple node
3. Clustered

Redis caching architectures are split across Azure by tiers:

- **Basic cache:** A basic cache provides you with a single node Redis cache. The complete dataset will be stored in a single node. This tier is ideal for development, testing, and non-critical workloads.
- **Standard cache:** The standard cache creates multiple node architectures. Redis replicates a cache in a two-node primary/secondary configuration. Azure manages the replication between the two nodes. This is a production-ready cache with master/slave replication.
- **Premium tier:** The premium tier includes the features of the standard tier but adds the ability to persist data, take snapshots, and back up data. With this tier, you can create a Redis cluster that shards data across multiple Redis nodes to increase available memory. The premium tier also supports an Azure Virtual Network to give you complete control over your connections, subnets, IP addressing, and network isolation. This tier also includes geo-replication, so you can ensure your data is close to the app that's consuming it.

Summary

A database is great for storing large amounts of data, but there is an inherent latency when looking up data. You send a query. The server interprets the query, looks up the data, and returns it. Servers also have capacity limits for handling requests. If too many requests are made, data retrieval will likely slow down. Caching will store frequently requested data in memory that can be returned faster than querying a database, which should lower latency and increase performance. Azure Cache for Redis gives you access to a secure, dedicated, and scalable Redis cache, hosted in Azure, and managed by Microsoft.

Configure Azure Cache for Redis

Create and configure the Azure Cache for Redis instance

You can create a Redis cache using the Azure portal, the Azure CLI, or Azure PowerShell. There are several parameters you will need to decide in order to configure the cache properly for your purposes.

Name

The Redis cache will need a globally unique name. The name has to be unique within Azure because it is used to generate a public-facing URL to connect and communicate with the service.

The name must be between 1 and 63 characters, composed of numbers, letters, and the '-' character. The cache name can't start or end with the '-' character, and consecutive '-' characters aren't valid.

Resource Group

The Azure Cache for Redis is a managed resource and needs a resource group owner. You can either create a new resource group, or use an existing one in a subscription you are part of.

Location

You will need to decide where the Redis cache will be physically located by selecting an Azure region. You should always place your cache instance and your application in the same region. Connecting to a cache in a different region can significantly increase latency and reduce reliability. If you are connecting to the cache outside of Azure, then select a location close to where the application consuming the data is running.

Important: Put the Redis cache as close to the data consumer as you can.

Pricing tier

As mentioned in the last unit, there are three pricing tiers available for an Azure Cache for Redis.

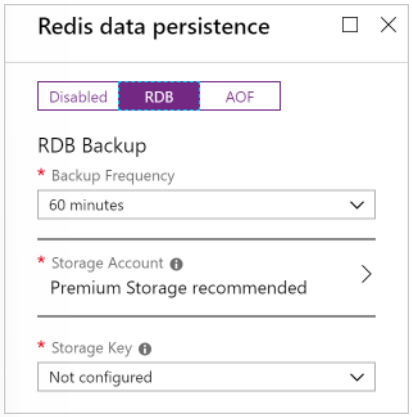
- **Basic:** Basic cache ideal for development/testing. Is limited to a single server, 53 GB of memory, and 20,000 connections. There is no SLA for this service tier.
- **Standard:** Production cache which supports replication and includes an 99.99% SLA. It supports two servers (master/slave), and has the same memory/connection limits as the Basic tier.
- **Premium:** Enterprise tier which builds on the Standard tier and includes persistence, clustering, and scale-out cache support. This is the highest performing tier with up to 530 GB of memory and 40,000 simultaneous connections.

You can control the amount of cache memory available on each tier - this is selected by choosing a cache level from C0-C6 for Basic/Standard and P0-P4 for Premium. Check the **pricing page**[1] for full details.

Tip: Microsoft recommends you always use Standard or Premium Tier for production systems. The Basic Tier is a single node system with no data replication and no SLA. Also, use at least a C1 cache. C0 caches are really meant for simple dev/test scenarios since they have a shared CPU core and very little memory.

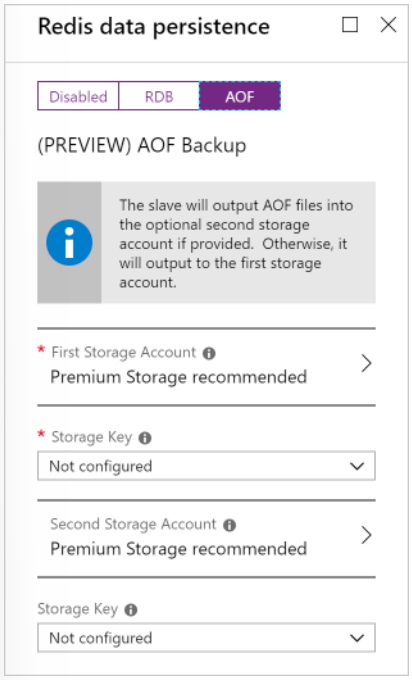
The Premium tier allows you to persist data in two ways to provide disaster recovery:

1. RDB persistence takes a periodic snapshot and can rebuild the cache using the snapshot in case of failure.



2.

3. AOF persistence saves every write operation to a log that is saved at least once per second. This creates bigger files than RDB but has less data loss.



4.

There are several other settings which are only available to the **Premium** tier.

1 https://azure.microsoft.com/pricing/details/cache/

Virtual Network support

If you create a premium tier Redis cache, you can deploy it to a virtual network in the cloud. Your cache will be available to only other virtual machines and applications in the same virtual network. This provides a higher level of security when your service and cache are both hosted in Azure, or are connected through an Azure virtual network VPN.

Clustering support

With a premium tier Redis cache, you can implement clustering to automatically split your dataset among multiple nodes. To implement clustering, you specify the number of shards to a maximum of 10. The cost incurred is the cost of the original node, multiplied by the number of shards.

Accessing the Redis instance

Redis supports a set of known commands. A command is typically issued as COMMAND parameter1 parameter2 parameter3.

Here are some common commands you can use:

Command	Description
`ping`	Ping the server. Returns "PONG".
`set [key] [value]`	Sets a key/value in the cache. Returns "OK" on success.
`get [key]`	Gets a value from the cache.
`exists [key]`	Returns '1' if the **key** exists in the cache, '0' if it doesn't.
`type [key]`	Returns the type associated to the value for the given **key**.
`incr [key]`	Increment the given value associated with **key** by '1'. The value must be an integer or double value. This returns the new value.
`incrby [key] [amount]`	Increment the given value associated with **key** by the specified amount. The value must be an integer or double value. This returns the new value.
`del [key]`	Deletes the value associated with the **key**.
`flushdb`	Delete *all* keys and values in the database.

Redis has a command-line tool (**redis-cli**) you can use to experiment directly with these commands. Here are some examples.

```
> set somekey somevalue
OK
> get somekey
"somevalue"
> exists somekey
(string) 1
> del somekey
(string) 1
> exists somekey
```

```
(string) 0
```

Here's an example of working with the `INCR` commands. These are convenient because they provide atomic increments across multiple applications that are using the cache.

```
> set counter 100
OK
> incr counter
(integer) 101
> incrby counter 50
(integer) 151
> type counter
(integer)
```

Adding an expiration time to values

Caching is important because it allows us to store commonly used values in memory. However, we also need a way to expire values when they are stale. In Redis this is done by applying a time to live (TTL) to a key.

When the TTL elapses, the key is automatically deleted, exactly as if the DEL command were issued. Here are some notes on TTL expirations.

- Expirations can be set using seconds or milliseconds precision.
- The expire time resolution is always 1 millisecond.
- Information about expires are replicated and persisted on disk, the time virtually passes when your Redis server remains stopped (this means that Redis saves the date at which a key will expire).

Here is an example of an expiration:

```
> set counter 100
OK
> expire counter 5
(integer) 1
> get counter
100
... wait ...
> get counter
(nil)
```

Accessing a Redis cache from a client

To connect to an Azure Cache for Redis instance, you'll need several pieces of information. Clients need the host name, port, and an access key for the cache. You can retrieve this information in the Azure portal through the **Settings > Access Keys** page.

- The host name is the public Internet address of your cache, which was created using the name of the cache. For example `sportsresults.redis.cache.windows.net`.
- The access key acts as a password for your cache. There are two keys created: primary and secondary. You can use either key, two are provided in case you need to change the primary key. You can switch all of your clients to the secondary key, and regenerate the primary key. This would block any applica-

tions using the original primary key. Microsoft recommends periodically regenerating the keys - much like you would your personal passwords.

Warning: Your access keys should be considered confidential information, treat them like you would a password. Anyone who has an access key can perform any operation on your cache!

Use the client API to interact with Redis

Redis is an in-memory NoSQL database which can be replicated across multiple servers. It is often used as a cache, but can be used as a formal database or even message-broker.

It can store a variety of data types and structures and supports a variety of commands you can issue to retrieve cached data or query information about the cache itself. The data you work with is always stored as key/value pairs.

Executing commands on the Redis cache

Typically, a client application will use a client library to form requests and execute commands on a Redis cache. You can get a list of client libraries directly from the Redis clients page. A popular high-performance Redis client for the .NET language is **StackExchange.Redis**. The package is available through NuGet and can be added to your .NET code using the command line or IDE.

Connecting to your Redis cache with StackExchange.Redis

Recall that we use the host address, port number, and an access key to connect to a Redis server. Azure also offers a connection string for some Redis clients which bundles this data together into a single string.

A connection string is a single line of text that includes all the required pieces of information to connect and authenticate to a Redis cache in Azure. It will look something like the following (with the `cache-name` and `password-here` fields filled in with real values):

```
[cache-name].redis.cache.windows.net:6380,password=[pass-
word-here],ssl=True,abortConnect=False
```

You can pass this string to **StackExchange.Redis** to create a connection the server.

Notice that there are two additional parameters at the end:

- **ssl** - ensures that communication is encrypted.
- **abortConnection** - allows a connection to be created even if the server is unavailable at that moment.

There are several other **optional parameters**[2] you can append to the string to configure the client library.

Tip: The connection string should be protected in your application. If the application is hosted on Azure, consider using an Azure Key Vault to store the value.

Creating a connection

The main connection object in **StackExchange.Redis** is the `StackExchange.Redis.ConnectionMultiplexer` class. This object abstracts the process of connecting to a Redis server (or group of

2 https://github.com/StackExchange/StackExchange.Redis/blob/master/docs/Configuration.md#configuration-options

servers). It's optimized to manage connections efficiently and intended to be kept around while you need access to the cache.

You create a `ConnectionMultiplexer` instance using the static `ConnectionMultiplexer.Connect` or `ConnectionMultiplexer.ConnectAsync` method, passing in either a connection string or a `ConfigurationOptions` object.

Here's a simple example:

```
using StackExchange.Redis;
...
var connectionString = "[cache-name].redis.cache.windows.net:6380,password=[password-here],ssl=True,abortConnect=False";
var redisConnection = ConnectionMultiplexer.Connect(connectionString);
    // ^^^ store and re-use this!!!
```

Once you have a `ConnectionMultiplexer`, there are 3 primary things you might want to do:

1. Access a Redis Database. This is what we will focus on here.
2. Make use of the publisher/subscript features of Redis. This is outside the scope of this module.
3. Access an individual server for maintenance or monitoring purposes.

Accessing a Redis database

The Redis database is represented by the `IDatabase` type. You can retrieve one using the `GetDatabase()` method:

```
IDatabase db = redisConnection.GetDatabase();
```

Tip: The object returned from `GetDatabase` is a lightweight object, and does not need to be stored. Only the `ConnectionMultiplexer` needs to be kept alive.

Once you have a `IDatabase` object, you can execute methods to interact with the cache. All methods have synchronous and asynchronous versions which return `Task` objects to make them compatible with the `async` and `await` keywords.

Here is an example of storing a key/value in the cache:

```
bool wasSet = db.StringSet("favorite:flavor", "i-love-rocky-road");
```

The `StringSet` method returns a `bool` indicating whether the value was set (`true`) or not (`false`). We can then retrieve the value with the `StringGet` method:

```
string value = db.StringGet("favorite:flavor");
Console.WriteLine(value); // displays: ""i-love-rocky-road""
```

Getting and Setting binary values

Recall that Redis keys and values are binary safe. These same methods can be used to store binary data. There are implicit conversion operators to work with `byte[]` types so you can work with the data naturally:

```
byte[] key = ...;
byte[] value = ...;

db.StringSet(key, value);


byte[] key = ...;
byte[] value = db.StringGet(key);
```

StackExchange.Redis represents keys using the `RedisKey` type. This class has implicit conversions to and from both `string` and `byte[]`, allowing both text and binary keys to be used without any complication. Values are represented by the `RedisValue`type. As with `RedisKey`, there are implicit conversions in place to allow you to pass `string` or `byte[]`.

Other common operations

The IDatabase interface includes several other methods to work with the Redis cache. There are methods to work with hashes, lists, sets, and ordered sets.

Here are some of the more common ones that work with single keys, you can **read the source code**[3] for the interface to see the full list.

Method	Description
`CreateBatch`	Creates a *group of operations* that will be sent to the server as a single unit, but not necessarily processed as a unit.
`CreateTransaction`	Creates a group of operations that will be sent to the server as a single unit *and* processed on the server as a single unit.
`KeyDelete`	Delete the key/value.
`KeyExists`	Returns whether the given key exists in cache.
`KeyExpire`	Sets a time-to-live (TTL) expiration on a key.
`KeyRename`	Renames a key.
`KeyTimeToLive`	Returns the TTL for a key.
`KeyType`	Returns the string representation of the type of the value stored at key. The different types that can be returned are: string, list, set, zset and hash.

Executing other commands

The `IDatabase` object has an `Execute` and `ExecuteAsync` method which can be used to pass textual commands to the Redis server. For example:

```
var result = db.Execute("ping");
Console.WriteLine(result.ToString()); // displays: "PONG"
```

3 https://github.com/StackExchange/StackExchange.Redis/blob/master/src/StackExchange.Redis/Interfaces/IDatabase.cs

The `Execute` and `ExecuteAsync` methods return a `RedisResult` object which is a data holder that includes two properties:

- `Type` which returns a `string` indicating the type of the result - "STRING", "INTEGER", etc.
- `IsNull` a true/false value to detect when the result is `null`.

You can then use `ToString()` on the `RedisResult` to get the actual return value.

You can use `Execute` to perform any supported commands - for example, we can get all the clients connected to the cache ("CLIENT LIST"):

```
var result = await db.ExecuteAsync("client", "list");
Console.WriteLine($"Type = {result.Type}\r\nResult = {result}");
```

This would output all the connected clients:

```
Type = BulkString
Result = id=9469 addr=16.183.122.154:54961 fd=18 name=DESKTOP-AAAAAA age=0
idle=0 flags=N db=0 sub=1 psub=0 multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0
omem=0 ow=0 owmem=0 events=r cmd=subscribe numops=5
id=9470 addr=16.183.122.155:54967 fd=13 name=DESKTOP-BBBBBB age=0 idle=0
flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-free=32768 obl=0 oll=0 omem=0
ow=0 owmem=0 events=r cmd=client numops=17
```

Storing more complex values

Redis is oriented around binary safe strings, but you can cache off object graphs by serializing them to a textual format - typically XML or JSON. For example, perhaps for our statistics, we have a `GameStats` object which looks like:

```
public class GameStat
{
    public string Id { get; set; }
    public string Sport { get; set; }
    public DateTimeOffset DatePlayed { get; set; }
    public string Game { get; set; }
    public IReadOnlyList<string> Teams { get; set; }
    public IReadOnlyList<(string team, int score)> Results { get; set; }

    public GameStat(string sport, DateTimeOffset datePlayed, string game,
string[] teams, IEnumerable<(string team, int score)> results)
    {
        Id = Guid.NewGuid().ToString();
        Sport = sport;
        DatePlayed = datePlayed;
        Game = game;
        Teams = teams.ToList();
        Results = results.ToList();
    }

    public override string ToString()
    {
```

```
        return $"{Sport} {Game} played on {DatePlayed.Date.ToShortDat-
eString()} - " +
               $"{String.Join(',', Teams)}\r\n\t" +
               $"{String.Join('\t', Results.Select(r => $"{r.team } - {r.
score}\r\n"))}";
    }
}
```

We could use the **Newtonsoft.Json** library to turn an instance of this object into a string:

```
var stat = new GameStat("Soccer", new DateTime(1950, 7, 16), "FIFA World
Cup",
                new[] { "Uruguay", "Brazil" },
                new[] { ("Uruguay", 2), ("Brazil", 1) });

string serializedValue = Newtonsoft.Json.JsonConvert.SerializeObject(stat);
bool added = db.StringSet("event:1950-world-cup", serializedValue);
```

We could retrieve it and turn it back into an object using the reverse process:

```
var result = db.StringGet("event:1950-world-cup");
var stat = Newtonsoft.Json.JsonConvert.DeserializeObject<GameStat>(result.
ToString());
Console.WriteLine(stat.Sport); // displays "Soccer"
```

Cleaning up the connection

Once you are done with the Redis connection, you can `Dispose` the `ConnectionMultiplexer`. This will close all connections and shutdown the communication to the server.

```
redisConnection.Dispose();
redisConnection = null;
```

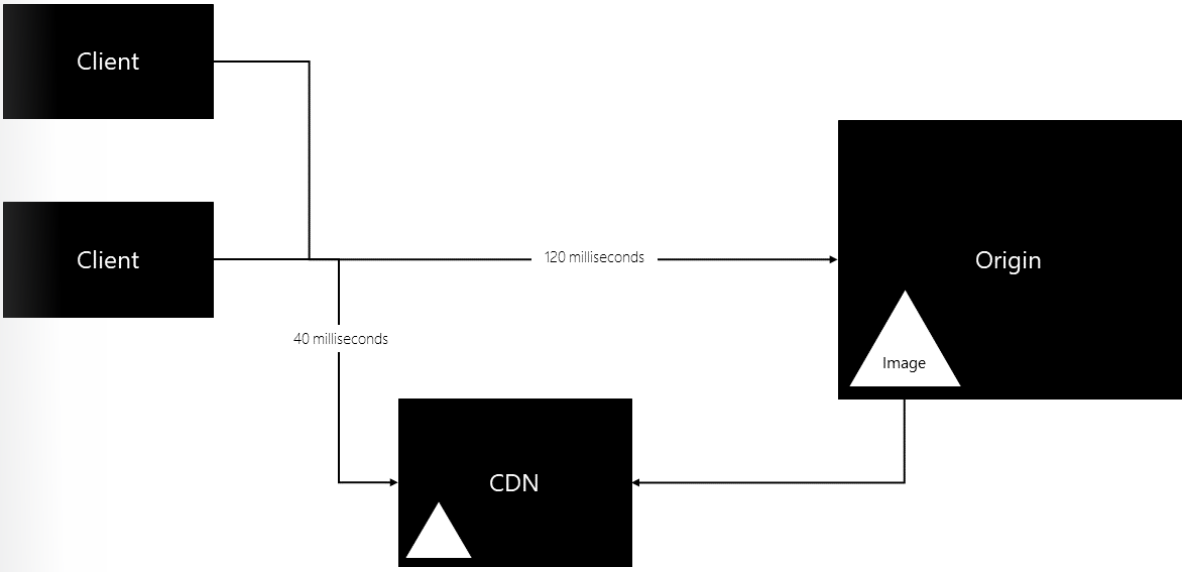
Redis limited user account (LUA) scripts

Develop for storage on CDNs

Content Delivery Network

A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users. CDNs store cached content on Edge servers that are close to users to minimize latency. These Edge servers are located in point of presence (POP) locations that are distributed throughout the globe.

CDNs are typically used to deliver static content, such as images, style sheets, documents, client-side scripts, and HTML pages. The major advantages of using a CDN are lower latency and faster delivery of content to users—regardless of their geographical location in relation to the datacenter where the application is hosted. CDNs can also help to reduce the load on a web application, because the application does not have to service requests for the content that is hosted in the CDN.



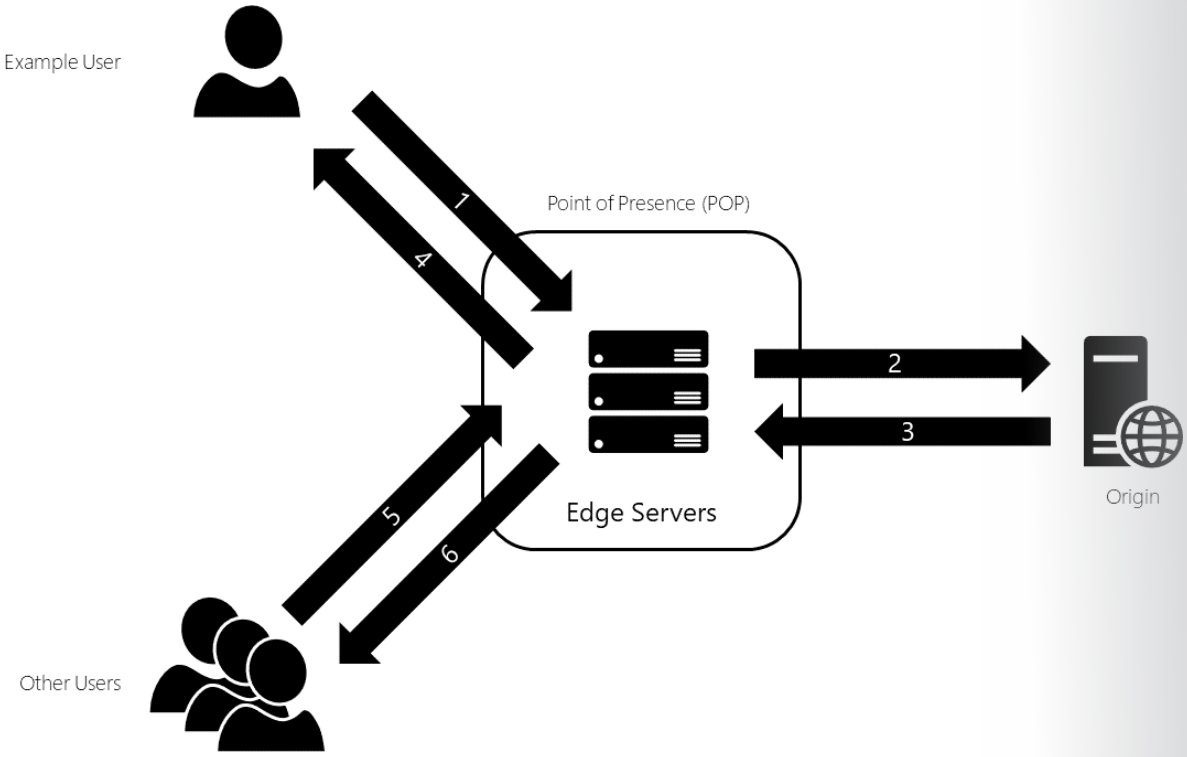
Typical uses for a CDN include:

- Delivering static resources, often from a website, for client applications. These resources can be images, style sheets, documents, files, client-side scripts, HTML pages, HTML fragments, or any other content that the server does not need to modify for each request.
- Delivering public static and shared content to devices such as mobile phones and tablets.
- Serving entire websites that consist of only public static content to clients without requiring any dedicated compute resources.
- Streaming video files to client devices on demand, taking advantage of the low latency and reliable connectivity available from the globally located datacenters that offer CDN connections.
- Supporting Internet of Things (IoT) solutions. The huge numbers of devices and appliances involved in an IoT solution can easily overwhelm an application if it has to distribute firmware updates directly to each device.
- Coping with peaks and surges in demand without requiring the application to scale, avoiding the consequent increased running costs.

Azure CDN

In Azure, the Azure Content Delivery Network (Azure CDN) is a global CDN solution for delivering high-bandwidth content that is hosted in Azure or in any other location. Using Azure CDN, you can cache publicly available objects loaded from Azure Blob storage, a web application, a virtual machine, or any publicly accessible web server. Azure CDN can also accelerate dynamic content, which cannot be cached, by taking advantage of various network optimizations by using CDN POPs. An example is using route optimization to bypass Border Gateway Protocol (BGP).

Here's how Azure CDN works.



1. A user (Example User) requests a file (also called an asset) by using a URL with a special domain name, such as `endpoint_name.azureedge.net`. This name can be an endpoint hostname or a custom domain. The Domain Name System (DNS) routes the request to the best-performing POP location, which is usually the POP that is geographically closest to the user.
2. If no Edge servers in the POP have the file in their cache, the POP requests the file from the origin server. The origin server can be an Azure web app, Azure cloud service, Azure storage account, or any publicly accessible web server.
3. The origin server returns the file to an Edge server in the POP.
4. An Edge server in the POP caches the file and returns the file to the original requestor (Example User). The file remains cached on the Edge server in the POP until the Time to Live (TTL) specified by its HTTP headers expires. If the origin server didn't specify a TTL, the default TTL is seven days.
5. Additional users can then request the same file by using the same URL that the original requestor (Example User) used, which can also be directed to the same POP.
6. If the TTL for the file hasn't expired, the POP Edge server returns the file directly from the cache. This process results in a faster, more responsive user experience.

Manage Azure CDN by using Azure CLI

The Azure Command-Line Interface (Azure CLI) provides one of the most flexible methods to manage your Azure CDN profiles and endpoints. You can get started by listing all of your existing CDN profiles:

```
az cdn profile list
```

This will globally list every CDN profile associated with your subscription. If you want to filter this list down to a specific resource group, you can use the `--resource-group` parameter:

```
az cdn profile list --resource-group ExampleGroup
```

To create a new profile, you should use the new **create** verb for the `az cdn profile` command group:

```
az cdn profile create --name DemoProfile --resource-group ExampleGroup
```

By default, the CDN will be created by using the standard tier and the Akamai provider. You can customize this further by using the `--sku` parameter and one of the following options:

- **Custom_Verizon**
- **Premium_Verizon**
- **Standard_Akamai**
- **Standard_ChinaCdn**
- **Standard_Verizon**

After you have created a new profile, you can use that profile to create an endpoint. Each endpoint requires you to specify a profile, a resource group, and an origin URL:

```
az cdn endpoint create --name ContosoEndpoint --origin www.contoso.com
--profile-name DemoProfile --resource-group ExampleGroup
```

You can customize the endpoint further by assigning a custom domain to the CDN endpoint. This helps ensure that users see only the domains you choose instead of the Azure CDN domains:

```
az cdn custom-domain create --name FilesDomain --hostname files.contoso.com
--endpoint-name ContosoEndpoint --profile-name DemoProfile --resource-group
ExampleGroup
```

Cache expiration in Azure CDN

Because a cached resource can potentially be out-of-date or stale (compared to the corresponding resource on the origin server), it is important for any caching mechanism to control when content is refreshed. To save time and bandwidth consumption, a cached resource is not compared to the version on the origin server every time it is accessed. Instead, as long as a cached resource is considered to be fresh, it is assumed to be the most current version and is sent directly to the client. A cached resource is considered to be fresh when its age is less than the age or period defined by a cache setting. For example, when a browser reloads a webpage, it verifies that each cached resource on your hard drive is fresh and loads it. If the resource is not fresh (stale), an up-to-date copy is loaded from the server.

Caching rules

Azure CDN caching rules specify cache expiration behavior both globally and with custom conditions. There are two types of caching rules:

- Global caching rules. You can set one global caching rule for each endpoint in your profile that affects all requests to the endpoint. The global caching rule overrides any HTTP cache-directive headers, if set.
- Custom caching rules. You can set one or more custom caching rules for each endpoint in your profile. Custom caching rules match specific paths and file extensions; are processed in order; and override the global caching rule, if set.

For global and custom caching rules, you can specify the cache expiration duration in days, hours, minutes, and seconds.

Purging and preloading assets by using the Azure CLI

The Azure CLI provides a special purge verb that will unpublish cached assets from an endpoint. This is very useful if you have an application scenario where a large amount of data is invalidated and should be updated in the cache. To unpublish assets, you must specify either a file path, a wildcard directory, or both:

```
az cdn endpoint purge --content-paths '/css/*' '/js/app.js' --name Conto-
soEndpoint --profile-name DemoProfile --resource-group ExampleGroup
```

You can also preload assets into an endpoint. This is useful for scenarios where your application creates a large number of assets, and you want to improve the user experience by prepopulating the cache before any actual requests occur:

```
az cdn endpoint load --content-paths '/img/*' '/js/module.js' --name Conto-
soEndpoint --profile-name DemoProfile --resource-group ExampleGroup
```

Review Questions

Module 4 review questions

Azure Redis Cache

Can you describe what the Azure Cache for Redis is and what its two tiers of service?

> Click to see suggested answer

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

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

- **Basic.** Includes a single node.
- **Standard.** Includes two nodes in the primary replica configuration and includes replication support and a Service Level Agreement (SLA).

Cache expiration in Azure CDN

Because a cached resource can potentially be out-of-date or stale (compared to the corresponding resource on the origin server), it is important for any caching mechanism to control when content is refreshed. To save time and bandwidth consumption, a cached resource is not compared to the version on the origin server every time it is accessed. Instead, as long as a cached resource is considered to be fresh, it is assumed to be the most current version and is sent directly to the client.

Can you describe the two types of caching rules?

> Click to see suggested answer

Azure CDN caching rules specify cache expiration behavior both globally and with custom conditions. There are two types of caching rules:

- Global caching rules. You can set one global caching rule for each endpoint in your profile that affects all requests to the endpoint. The global caching rule overrides any HTTP cache-directive headers, if set.
- Custom caching rules. You can set one or more custom caching rules for each endpoint in your profile. Custom caching rules match specific paths and file extensions; are processed in order; and override the global caching rule, if set.

For global and custom caching rules, you can specify the cache expiration duration in days, hours, minutes, and seconds.