

Manage Azure PaaS tasks using automation

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/1-introduction>

Introduction

Completed

If you're proficient in SQL Server, you're likely familiar with the capabilities of the [SQL Server Agent](#). SQL Server Agent is also available on Azure SQL Managed Instance. However, if you're using Azure SQL Database, you'll need an alternative scheduling mechanism, as both the `msdb` database and the SQL Server Agent are unavailable.

In this module you'll learn about Azure Automation, Logic Apps, and elastic jobs, three approaches for automating jobs in PaaS.

You'll evaluate different strategies for monitoring your automation tasks. This includes setting up alerts and notifications to stay informed about job statuses and system errors, using performance metrics to track the effectiveness of your automation, and employing tools to troubleshoot and optimize your automated processes

2. Explore Elastic jobs

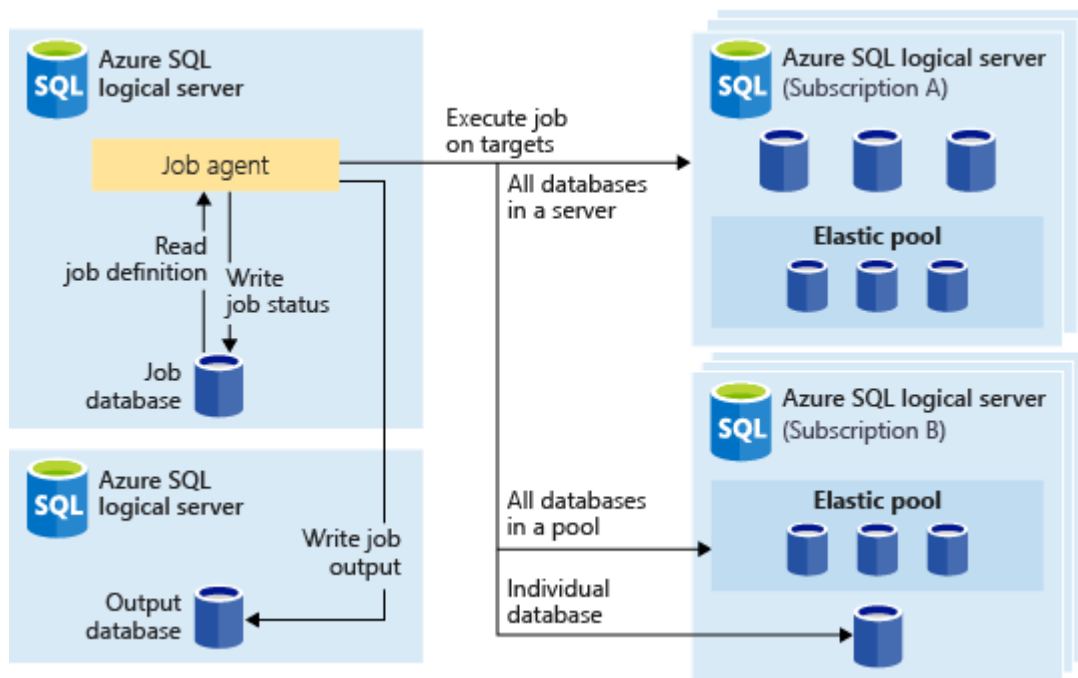
<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/2-explore-elastic-jobs>

Explore Elastic jobs

Completed

Many DBA's became familiar with Azure Automation because Azure SQL Database initially lacked capabilities for scheduled jobs.

The elastic jobs capability allows you to run a set of T-SQL scripts against a collection of servers or databases either as a one-time job or on a defined schedule. Elastic Jobs function similarly to SQL Server Agent jobs but are limited to executing T-SQL. They work across all tiers of Azure SQL Database. SQL Agent jobs continue to be used for task automation in SQL Server and are also included with Azure SQL Managed Instances.



To configure Elastic Jobs, you need a Job Agent and a dedicated database to manage your jobs. The recommended service tier for the job database is S1 or higher, depending on the number and frequency of jobs you're executing.

Let's review the components of Elastic Jobs:

- **Elastic job agent:** Your Azure resource for running and managing jobs.
- **Job database:** A dedicated database to manage your jobs.
- **Target group:** A collection of servers, elastic pools, and single databases where a job will be run.
- **Job:** One or more T-SQL scripts that make up a job step.

If a server or elastic pool is the target, you need to create a credential within the `master` database of the server or pool so the job agent can enumerate the databases. For a single database, only a database credential is needed. Credentials should have the least privileges necessary to perform the job step.

Elastic Job agent ...

Microsoft

Basics Review + create

An Elastic Job agent runs jobs whose definitions are stored in an Azure SQL Database. A job is a T-SQL script that is scheduled or executed ad-hoc against a group of Azure SQL databases. [Learn more](#) 

Name *

ElasticJobAgent ✓

Subscription * ⓘ

Contoso Subscription ▼

Job database

JobSQLDatabase (contoso-east-jp, eastus)

[Select Job database](#)

Review + create

Next : Review + create >

You can create an elastic job agent through the Azure portal, PowerShell, or REST API. To create an Elastic Job agent from the Azure portal, navigate to the **Elastic Job Agent** page. Here, provide a name for your agent and specify an SQL database for your job database.

You can create a target group using PowerShell, T-SQL, or Azure CLI. The following snippet creates a *MyServerGroup* target group, including all databases on the server at the time of execution. This snippet assumes the variables `$jobAgent` and `$targetServerName` were previously provided.

```
# create MyServerGroup target group
$serverGroup = $jobAgent | New-AzSqlElasticJobTargetGroup -Name 'MyServerGroup'
$serverGroup | Add-AzSqlElasticJobTarget -ServerName $targetServerName -RefreshCred
```

This script sets up an elastic job target group and adds a target server to it, allowing you to run jobs against the specified server.

The following code snippet creates an elastic job and adds job steps using PowerShell. *Step1* is responsible for creating the *MyTable* table if it doesn't already exist.

```
Write-Output "Creating a new job..."
$jobName = "MyFirstElasticJob"
$job = $jobAgent | New-AzSqlElasticJob -Name $jobName -RunOnce

Write-Output "Creating job steps for $($jobName) job..."
```

```
$sqlText1 = "IF NOT EXISTS (SELECT * FROM sys.tables WHERE object_id = object_id('I
$job | Add-AzSqlElasticJobStep -Name "Step1" -TargetGroupName $serverGroup.TargetGr
```

The T-SQL scripts executed by elastic jobs should be idempotent. This means that if the job runs multiple times, whether accidentally or due to a job failure, it won't fail or produce unintended results. You should be able to run the same script multiple times without encountering errors.

Finally, run the elastic job *MyFirstElasticJob* using PowerShell.

```
Write-Output "Start the job..."
$jobExecution = $job | Start-AzSqlElasticJob
$jobExecution
```

Use case scenarios

Elastic jobs can be used in the following scenarios:

- Automate management tasks to run on a specific schedule.
- Deploy schema changes seamlessly.
- Move data efficiently.
- Collect and aggregate data for reporting or other purposes.
- Load data from Azure Blob storage.
- Configure jobs to execute across multiple databases on a recurring basis, such as during off-peak hours.
- Process data across numerous databases, such as telemetry collection, and compile results into a single destination table for further analysis.

SQL Agent jobs and Elastic Jobs serve different purposes across Azure SQL platforms. SQL Agent jobs, available in SQL Server and Azure SQL Managed Instances, offer robust scheduling and automation. Elastic Jobs, designed for Azure SQL Database and elastic pools, run T-SQL scripts across multiple databases.

3. Understand Azure Automation

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/3-understand-azure-automation>

Understand Azure Automation

Completed

Azure offers several ways to automate processes. Azure Functions and Logic Apps are services that enable serverless workloads, creating workflows composed of steps to execute complex tasks. For example, a Logic App can populate a table in an Azure SQL Database when an entry is made in a SharePoint list. A full explanation of these services is beyond the scope of this course.

For more control and granularity in your automation, Azure Automation provides process automation, configuration management, full integration with Azure platform options (such as role-based access control and Microsoft Entra ID), and the ability to manage both Azure and on-premises resources.

One unique benefit of Azure Automation is its capability to manage resources within Azure or on-premises VMs. For instance, if you have a VM that is kept in a down state for cost savings, you can use Azure Automation with hybrid runbooks to start the VM, perform a SQL Server backup, and then shut down the VM.

Another common use of Azure Automation is for periodic maintenance operations, such as purging stale or old data or reindexing an SQL database.

Azure Automation components

[Azure Automation](#) supports both automation and configuration management activities. While we'll focus on the automation components, Azure Automation can also manage server updates and desired state configuration. Here are the key components you need to execute automated tasks.

- **Runbooks:** Runbooks are the units of execution in Azure Automation. They can be defined as graphical runbooks based on PowerShell, PowerShell scripts, or Python scripts. PowerShell runbooks are most commonly used to manage Azure SQL resources.
- **Modules:** Azure Automation provides an execution context for the PowerShell or Python code in your runbook. To execute your code, you need to import the necessary modules. For example, to run the `Get-AzSqlDatabase` PowerShell cmdlet, you would import the `Az.SQL` PowerShell module into your automation account.
- **Credentials:** Credentials store sensitive information that runbooks or configurations can use at runtime.
- **Schedules:** Schedules are linked to runbooks and trigger a runbook at a specific time.

Azure policy

Group Policies (GPOs) have long been used by Windows server administrators to manage security and ensure consistency across Windows Server environments. Examples include enforcing password complexity and mapping shared network drives.

[Azure Policy](#) offers similar governance for Azure resources, enforcing rules like region limitations, naming standards, and resource sizes. Policies can be applied at various levels, such as management

groups, subscriptions, or resource groups. They can also be grouped into initiatives for broader application.

Another use of Azure Policy is resource tagging, which stores metadata in key-value pairs. For example, a policy might require all resources to have tags for environment and cost center, blocking deployment if tags are missing.

Azure subscriptions and tags

Organizations use multiple subscriptions for various reasons, such as budget management, security, or resource isolation. For instance, an organization might separate internal and customer-facing resources into different subscriptions to simplify billing and isolate internal resources. These subscriptions can be managed together in a management group, allowing for unified policy and compliance management across subscriptions.

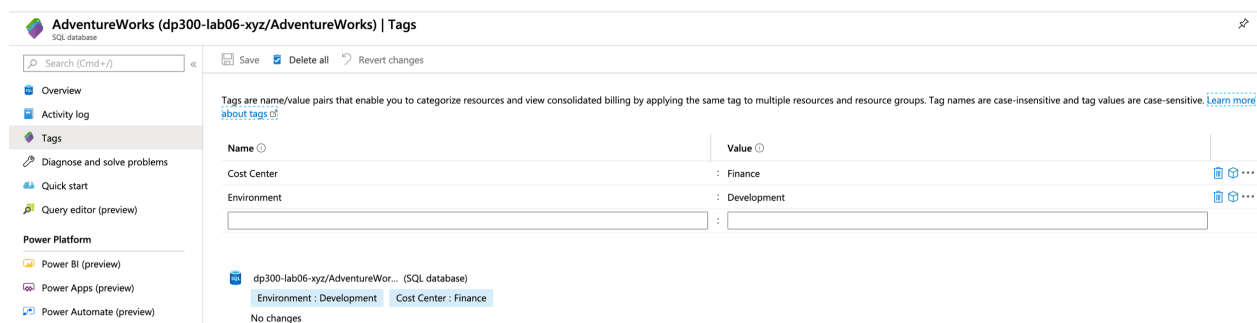
Tags are metadata used to describe your Azure resources better. Stored as **key:value** pairs, tags appear in the Azure portal alongside your resources. Because they're associated with the resource, you can filter PowerShell or Azure CLI commands based on tags, similar to using a **WHERE** clause in a SQL query. Here's a basic example:

```
$rg=(get-AzResourceGroup)

$rg=($rg|where-object {($_.tags['Use'] -ne 'Internal')}).ResourceGroupName
```

In the second line of this code sample, the list of resource groups is filtered by the tag called *'Use'*, returning only those resource groups where the tag value isn't *'Internal'*. Tags can be applied in the Azure portal, programmatically via PowerShell, Azure CLI, or as part of your deployment process. They can be applied at the subscription, resource group, or individual resource level and can be modified at any time. Azure supports up to 15 tags per resource.

Tags are also included in Azure billing information, making it easier for management to break down charges by cost center. Tags are found in the overview section of the blade for every Azure resource. To add tags to a resource using the Azure portal, select **Tags**, enter the key and value for your tag, and then select **Save**.



AdventureWorks (dp300-lab06-xyz/AdventureWorks) | Tags

Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. Tag names are case-insensitive and tag values are case-sensitive. [Learn more](#)

Name	Value
Cost Center	Finance
Environment	Development

dp300-lab06-xyz/AdventureWor... (SQL database)

Environment : Development Cost Center : Finance

No changes

The following example shows how to add tags using PowerShell.

```
$tags = @{"Dept"="Finance"; "Status"="Normal"}
```

```
$resource = Get-AzResource -Name demoStorage -ResourceGroup demoGroup  
New-AzTag -ResourceId $resource.id -Tag $tags
```

The following example shows how to add tags using Azure CLI.

```
az resource tag --tags 'Dept=IT' 'Environment=Test' -g examplegroup -n examplevnet  
--resource-type "Microsoft.Network/virtualNetworks"
```

Tags enable customers to organize Azure resources and management hierarchy into a taxonomy.

Note

Tags are stored as plain text, so never add sensitive values to them. Sensitive information could be exposed through various methods, including cost reports, tag taxonomies, deployment histories, exported templates, and monitoring logs.

4. Build an automation runbook

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/4-build-automation-runbook>

Build an automation runbook

Completed

Let's look at the steps required to create an automation account and an automation runbook.

Create an automation account

To create an automation runbook, you first need to create an automation account.

Create an Automation Account ...

Basics Advanced Networking Tags Review + Create

Create an Automation Account to hold the Automation runbooks & configuration used for automating operations and management tasks around Azure and non-Azure resources. You could execute cloud jobs in a serverless environment or use hybrid jobs on your compute via Azure Virtual machines or Arc-enabled servers. [Learn more](#)

Subscription * ⓘ

 Resource group * ⓘ
[Create new](#)

Instance Details

Automation account name * ⓘ

Region * ⓘ

[Review + Create](#)

[Previous](#)

[Next](#)

In this example, you connect to an Azure SQL Database using PowerShell. This requires importing modules to support the necessary cmdlets. Before creating your runbook, import the required modules into your Azure Automation account. Navigate to the Shared Resources section of the main blade for your automation account and select **Modules**. Import the **Az.Accounts** module, as the **Az.SQL** module depends on it.

Search for the module in the gallery as shown above. After selecting the module, select **Import**, as shown below. This imports the module into your account. Repeat this process for the **Az.SQL** module.

Next, you can optionally create a credential for your runbook to use. Create a credential by selecting on **Credentials** in the **Shared Resources** section of the main blade of your automation account. Creating a credential isn't mandatory for using Azure Automation, but this example includes one.



New Credential ×

Name *

 ✓

Description

User name *

 ✓

Password *

 ✓

Confirm password *

 ✓

Create a runbook

Automation executes your [runbooks](#) based on the logic defined inside them. If a runbook is interrupted, it restarts at the beginning. This behavior requires you to write runbooks that support being restarted if transient issues occur.

On the **Process Automation** section of your Automation Account, select **Runbooks**, to create a runbook.



Create a runbook ...

Name *	DemoDP300
Runbook type *	PowerShell
Runtime version *	7.1 (preview)
Description	

i During runbook execution, PowerShell modules targeting 7.1 runtime version will be used. Please make sure the required PowerShell modules are present in 7.1 runtime version.

Create

Cancel

To create a runbook, you need to provide a name, the type of runbook, the runtime version, and optionally a description. Since this example specifies PowerShell as the type, a PowerShell editor opens.

On the editor, provide the following PowerShell code. We use system-managed identity to log in to Azure. You need to enable appropriate RBAC permissions for the system identity of this Automation account. Otherwise, the runbook might fail.

```
# Uses the system-managed identity to log in to Azure
try {
    Write-Output "Logging in to Azure..."
    Connect-AzAccount -Identity
} catch {
    Write-Error -Message $_.Exception
    throw $_.Exception
}

# Get SQL Database name
$dbname = (Get-AzSQLDatabase -ResourceGroupName 'SQLDB' -ServerName 'GSData' -Data)

# Set SQL Server name
$AzureSQLServerName = $dbname + ".database.windows.net"

# Get SQL credentials
```

```
$Cred = Get-AutomationPSCredential -Name "SQLUser"

# Execute SQL query
$SQLOutput = $(Invoke-Sqlcmd -ServerInstance $AzureSQLServerName -Username $Cred.UserName -Query "SELECT * FROM [dbo].[Table1]")

Write-Output $SQLOutput
```

In this example, we use system-managed identity to log in to Azure, retrieves information about an Azure SQL Database, runs a query, and returns the results.

You can select **Test pane** in the code editor in the Azure portal. This allows you to test your code in the context of Azure Automation. A typical development process involves creating your PowerShell code locally and then testing it within the automation environment. This helps separate any PowerShell errors from those generated by the automation context. Always test your code within automation to ensure there are no errors in the code itself.

5. Automate database workflows by using Logic Apps

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/5-automate-database-workflows-by-using-logic-apps>

Automate database workflows by using Logic Apps

Completed

Azure Logic Apps is a cloud-based platform designed to create and run automated workflows that integrate your apps, data, services, and systems. This platform enables you to quickly develop highly scalable integration solutions for both enterprise and business-to-business (B2B) scenarios. As part of Azure Integration Services, Azure Logic Apps simplifies connecting legacy, modern, and cutting-edge systems across cloud, on-premises, and hybrid environments.

Here are some example tasks, business processes, and workloads you can automate using Azure Logic Apps:

- Schedule and send email notifications using Office 365 when specific events occur, such as a new file upload.
- Route and process customer orders across on-premises systems and cloud services.
- Move uploaded files from an SFTP or FTP server to Azure Storage.
- Monitor tweets, analyze sentiment, and create alerts or tasks for items that need review.

Why Use Azure Logic Apps?

Azure Logic Apps provides prebuilt, Microsoft-managed API connectors and built-in operations, making it easier and quicker to connect and integrate apps, data, services, and systems. This allows you to focus on designing and implementing your solution's business logic and functionality, rather than figuring out how to access your resources.

Typically, you won't need to write any code. However, if you do, you can create code snippets using Azure Functions and run them from your workflow. You can also use the Inline Code action to run code snippets directly within your workflow. If your workflow needs to interact with events from Azure services, custom apps, or other solutions, you can monitor, route, and publish events using Azure Event Grid.

Logic Apps is fully managed by Microsoft Azure, freeing you from concerns about hosting, scaling, managing, monitoring, and maintaining solutions built with these services. By using these capabilities to create "serverless" apps and solutions, you can focus solely on the business logic and functionality. These services automatically scale to meet your needs, speed up integrations, and help you build robust cloud apps with little to no code.

SQL Server Connector

The SQL Server connector in Azure Logic Apps allows you to access your SQL database and create automated workflows triggered by events in your SQL database or other systems. This enables you to manage your SQL data and resources efficiently.

For example, you can use actions to get, insert, and delete data, as well as run SQL queries and stored procedures. You can create a workflow that checks for new records in a non-SQL database, processes the data, creates new records in your SQL database, and sends email alerts about the new records.

The SQL Server connector supports the following SQL editions:

- SQL Server
- Azure SQL Database
- Azure SQL Managed Instance

The SQL Server connector requires that your tables contain data so that SQL connector operations can return results when called. For instance, if you use Azure SQL Database, you can use the included sample databases to try out the SQL connector operations.

For an SQL database in Azure, the connection string has the following format:

```
Server=tcp:{server-name}.database.windows.net,1433;Initial Catalog={database-name}
```

Alternatively, you can check the connection string for your Azure SQL Database in the Azure portal. In the **Overview** section for your database, select **Show database connection strings** under the

Connection strings property.

If you want to start your workflow with a SQL Server trigger operation, you need to begin with a blank workflow.

The SQL Server connector is available for logic app workflows in multitenant Azure Logic Apps, integration service environment (ISE), and single-tenant Azure Logic Apps:

- **Consumption workflows in multi-tenant Azure Logic Apps:** This connector is available only as a managed connector. For more information, review the [managed SQL Server connector operations](#) page.
- **Consumption workflows in an integration service environment:** This connector is available as both a managed connector and an ISE connector designed to run in an ISE. For more information, review the managed SQL Server connector operations.
- **Standard workflows in single-tenant Azure Logic Apps:** This connector is available as both a managed connector and a built-in connector designed to run in the same process as the single-tenant Azure Logic Apps runtime. However, the built-in version differs in the following ways:
 - The built-in SQL Server connector has no triggers.
 - The built-in SQL Server connector has only one operation: Execute Query.

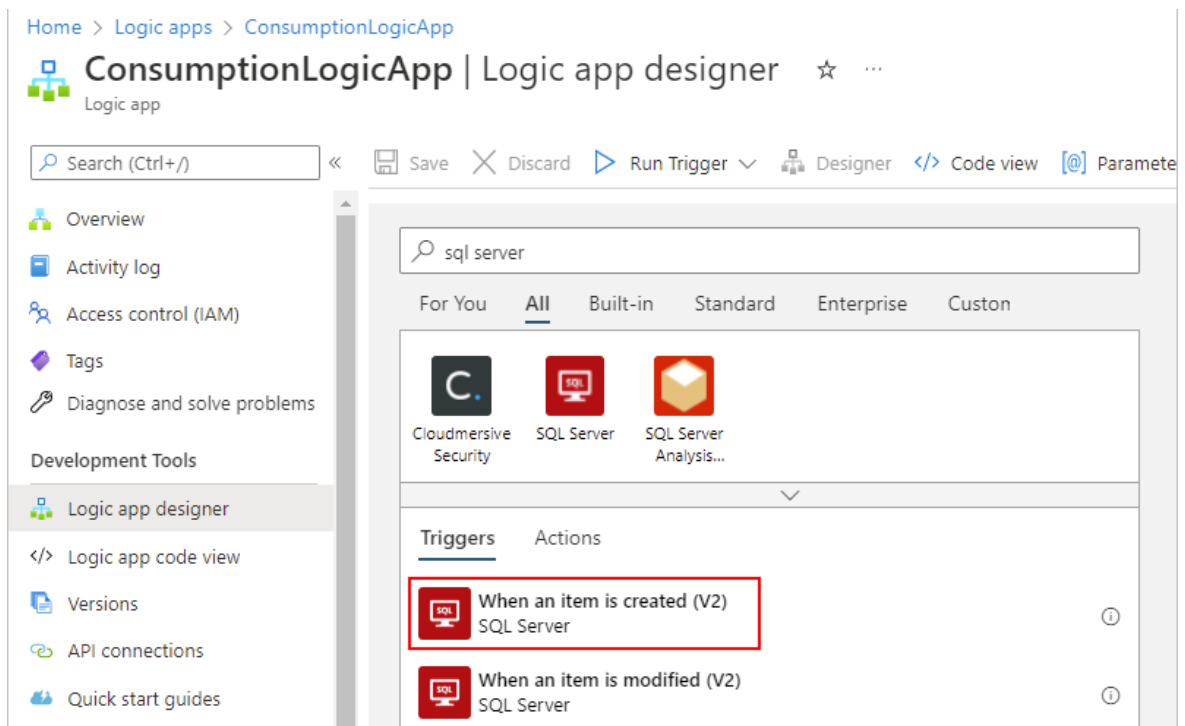
Create a logic app workflow

The following steps use the Azure portal to create logic app workflow.

Add a SQL Server trigger

The following steps use the Azure portal, but with the appropriate Azure Logic Apps extension, you can also use Visual Studio Code to create logic app workflows:

1. In the Azure portal, open your blank logic app workflow in the designer.
2. Find and select the managed SQL Server connector trigger that you want to use. Under the designer search box, select **All**.
3. In the designer search box, enter *sql server*.
4. From the triggers list, select the SQL trigger that you want. This example uses the trigger named **When an item is created**.



5. If you're connecting to your SQL database for the first time, you're prompted to create your SQL database connection now. After you create this connection, you can continue with the next step.
6. In the trigger properties, specify the interval and frequency for how often the trigger checks the table.
7. To add other properties available for this trigger, open the **Add new parameter** list and select those properties.

Note

This trigger returns only one row from the selected table, and nothing else. To perform other tasks, continue by adding either a SQL Server connector action or >another action that performs the next task that you want in your logic app workflow.

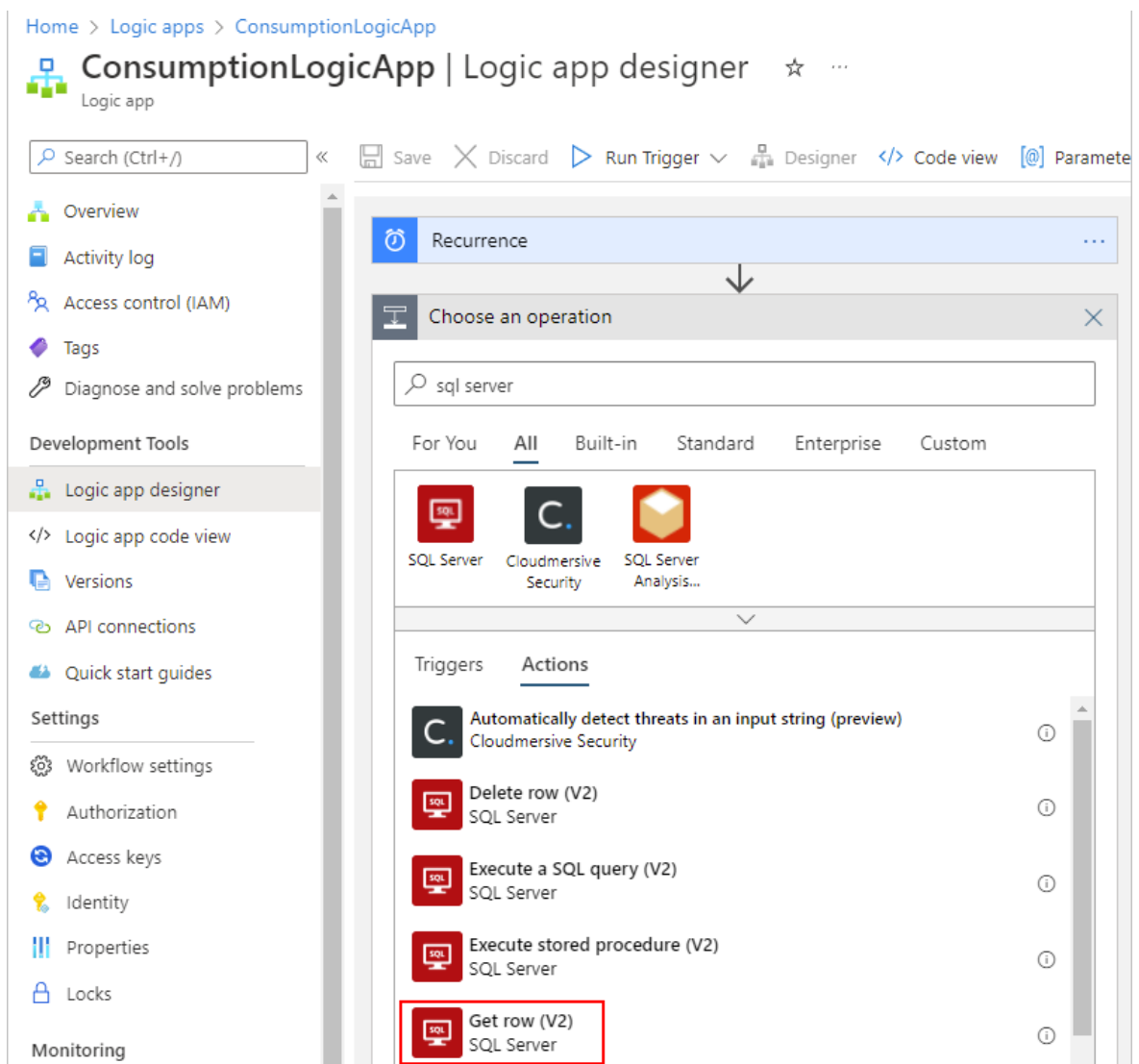
For example, to view the data in this row, you can add other actions that create a file that includes the fields from the returned row, and then send email alerts. To learn about other available actions for this connector, see the connector's reference page.

8. On the designer toolbar, select **Save**. this step automatically enables and publishes your logic app live in Azure

Add a SQL Server action

The following steps use the Azure portal. In this example, the logic app workflow starts with the Recurrence trigger, and calls an action that gets a row from an SQL database.

1. In the Azure portal, open your logic app workflow in the designer.
2. Find and select the managed SQL Server connector action that you want to use. This example uses the action named **Get row**.
3. Under the trigger or action where you want to add the SQL action, select **New step**.
4. In the **Choose an operation** box, under the designer search box, select **All**.
5. In the designer search box, enter *sql server*.
6. From the actions list, select the SQL Server action that you want. This example uses the **Get row** action, which gets a single record.



7. If you haven't already provided the SQL server name and database name, provide those values. Otherwise, from the Table name list, select the table that you want to use. In the *Row ID* property, enter the ID for the record that you want. In this example, the table name is *SalesLT.Product*.

The screenshot shows a workflow designer interface. At the top, there is a blue bar labeled 'Recurrence' with a clock icon and a three-dot menu. An arrow points down from this bar to a red bar labeled 'Get row (V2)' with a database icon and a three-dot menu. Below the 'Get row (V2)' bar, there are two input fields: '* Table name' with a dropdown menu showing 'SalesLT.Product', and '* Row id' with a text box containing 'Unique identifier of the row to retrieve'. Below these fields, a status bar indicates 'Connected to MySQLServerConnection.' with a 'Change connection.' link. At the bottom of the workflow area, there is a '+ New step' button.

Note

This action returns only one row from the selected table, and nothing else.

8. When you're done, on the designer toolbar, select **Save**.

Connect to Azure SQL Database

In the workflow designer, you must create a connection the first time you add a trigger or action for the first time. This information varies depending on the connection, for example:

- The name that you want to use for the new connection
- The name for the system or server
- Your user or account credentials
- The authentication type to use

6. Monitor automated tasks

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/6-monitor-automated-tasks>

Monitor automated tasks

Completed

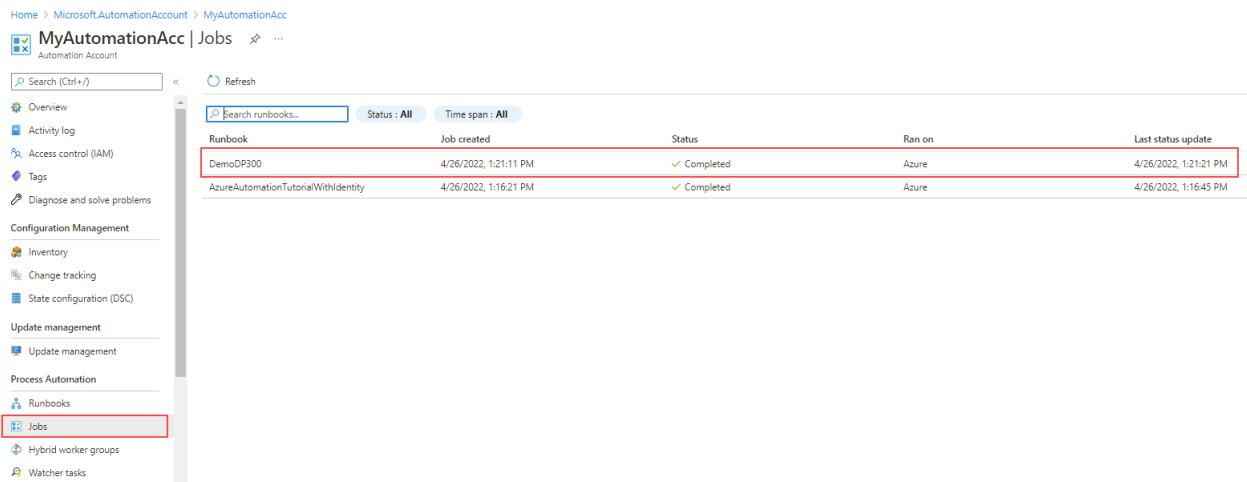
After automating your tasks, it's crucial to monitor them to ensure they're functioning correctly. This helps maximize the performance and availability of your services and proactively identify problems.

Monitor Runbooks

Your runbooks should be modular, with reusable, and easily restartable logic. Monitoring a runbook's progress ensures that its logic is executed correctly if issues arise.

You can use a database, storage account, or shared file to monitor a runbook's progress. Ensure your runbook checks the last action performed before initiating the next action. Based on the results, the logic can either skip or continue specific tasks in the runbook.

You can monitor runbook execution in the Azure portal. Select **Job** in the **Process Automation** section of the main blade of your automation account.



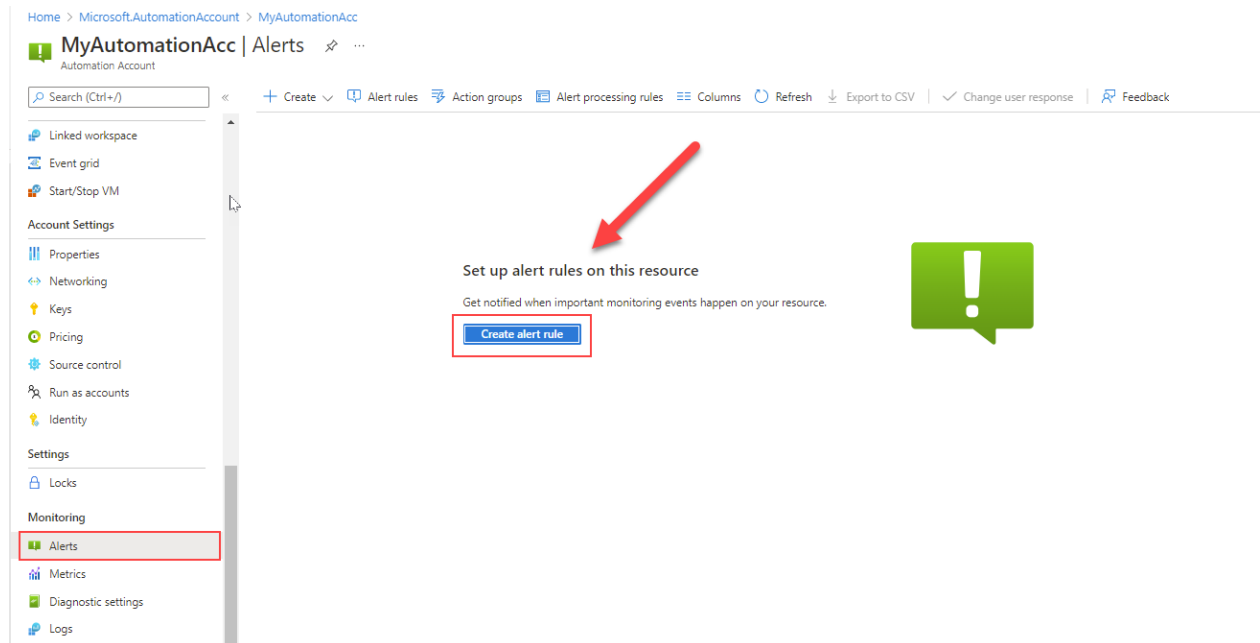
Runbook	Job created	Status	Ran on	Last status update
DemoDP300	4/26/2022, 1:21:11 PM	✓ Completed	Azure	4/26/2022, 1:21:21 PM
AzureAutomationTutorialWithIdentity	4/26/2022, 1:16:21 PM	✓ Completed	Azure	4/26/2022, 1:16:45 PM

As shown in the following image, the highlighted runbook was completed. To investigate the details and status, select the job.

To learn more about how to troubleshoot runbooks problems, see [Troubleshoot runbook issues](#).

Alerts

You can also create metric alerts to monitor the execution of your runbooks. Alerts allow you to define conditions to monitor and actions to take when those conditions are met.



When you select **Create alert rule**, the **Select a signal** slide-out opens on the right side of your screen. Next, select a signal that is most appropriate for your scenario. For this example, select **Total Jobs**.

Select a signal



Choose a signal below and configure the logic on the next screen to define the alert condition.

Signal type ⓘ

All

Monitor service ⓘ

All

Displaying 1 - 10 signals out of total 10 signals

🔍 Search by signal name

Signal name	↑↓	Signal type	↑↓	Monitor service	↑↓
Custom log search		Log search		Log analytics	
Total Jobs		Metrics		Platform	
Total Update Deployment Machine Runs		Metrics		Platform	
Total Update Deployment Runs		Metrics		Platform	
All Administrative operations		Activity log		Administrative	
Convert Graph Runbook Content to its raw serialized format and vice-versa (Micr...		Activity log		Administrative	
Generate a URI for an Azure Automation webhook (Microsoft.Automation/autom...		Activity log		Administrative	
Create or Update an Azure Automation account (Microsoft.Automation/automati...		Activity log		Administrative	
Gets the Keys for the automation account (Microsoft.Automation/automationAcc...		Activity log		Administrative	
Delete an Azure Automation account (Microsoft.Automation/automationAccounts)		Activity log		Administrative	

Done

In the **Configure signal logic** slide-out, select **Static** for the **Threshold** property. Then, set the **Operator** property to **Greater than**, and the **Aggregation** type to **Total**. Enter **10** for the **Threshold value**.

Configure signal logic



Split by dimensions

Use dimensions to monitor specific time series and provide context to the fired alert. Dimensions can be either number or string columns. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. [About monitoring multiple time series](#) ⓘ

Dimension name	Operator	Dimension values
Select dimension ▼	= ▼	0 selected ▼ Add custom value

Monitoring 1 time series (\$0.1/time series)

Alert logic

Threshold ⓘ

Static Dynamic

Operator ⓘ	Aggregation type * ⓘ	Threshold value * ⓘ	Unit * ⓘ
Greater than ▼	Total ▼	10 ✓	Count ▼

Condition preview

Whenever the total total jobs is greater than 10

Evaluated based on

Aggregation granularity (Period) * ⓘ 5 minutes ▼	Frequency of evaluation ⓘ Every 5 Minutes ▼
---	--

Done

Alternatively, you can specify a dimension. For example, you can define that an alert rule will only trigger for a specific runbook and status. If you don't specify a dimension, no filter is applied.

Split by dimensions

Use dimensions to monitor specific time series and provide context to the fired alert. Dimensions can be either number or string columns. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. [About monitoring multiple time series](#) ⓘ

Alert logic

Next, configure an action group. An action group is a collection of actions that you can use across multiple alerts, including email notifications, runbooks, webhooks, and more.

[NOTE] You can combine Azure Automation with Azure Alerts action groups to initiate an Automation runbook when an alert is raised.

Activity log

In Azure Automation, runbooks are executed and details are collected in an activity log.

Operation name	Status	Time	Time stamp	Subscription	Event initiated by
> Publish an Azure Automation runbook draft	Succeeded	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Create an Azure Automation job	Accepted	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Write an Azure Automation runbook draft	Succeeded	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Write an Azure Automation runbook draft	Succeeded	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Publish an Azure Automation runbook draft	Succeeded	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Write an Azure Automation runbook draft	Succeeded	38 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Publish an Azure Automation runbook draft	Accepted	39 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Create an Azure Automation job	Accepted	43 minutes ...	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Create or Update an Azure Automation schedule asset	Succeeded	2 hours ago	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Create or Update an Azure Automation Runbook	Succeeded	2 hours ago	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Create or Update an Azure Automation Runbook	Succeeded	2 hours ago	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com
> Write an Azure Automation runbook draft	Succeeded	2 hours ago	Tue Apr 26 ...	Contoso Subscription	contoso@contoso.com

In the following image, you can retrieve runbook details, such as the person or account that started a runbook.

As an alternative, the following PowerShell example provides the last user to run the specified runbook.

```

$rgName = 'MyResourceGroup'
$accountName = 'MyAutomationAccount'
$runbookName = 'MyRunbook'
$startTime = (Get-Date).AddDays(-1)

$params = @{
    ResourceGroupName = $rgName
    StartTime         = $startTime
}
$JobActivityLogs = (Get-AzLog @params).Where( { $_.Authorization.Action -eq 'Micros

$JobInfo = @{}
foreach ($log in $JobActivityLogs) {
    # Get job resource
    $JobResource = Get-AzResource -ResourceId $log.ResourceId

    if ($null -eq $JobInfo[$log.SubmissionTimestamp] -and $JobResource.Properties.F
        # Get runbook
        $jobParams = @{
            ResourceGroupName      = $rgName
            AutomationAccountName = $accountName
            Id                     = $JobResource.Properties.JobId
        }
        $Runbook = Get-AzAutomationJob @jobParams | Where-Object RunbookName -EQ $r

        # Add job information to hashtable
        $JobInfo.Add($log.SubmissionTimestamp, @($Runbook.RunbookName, $Log.Caller
    }
}
$JobInfo.GetEnumerator() | Sort-Object Key -Descending | Select-Object -First 1

```

Log Analytics

Azure Automation can send runbook job status and job streams to your Log Analytics workspace. This method doesn't require workspace linking and is independent.

Azure Monitor logs integrated with Automation account, enables you to:

- View the status of your Automation jobs
- Write advanced queries across your job workflow
- Trigger an email or alert based on your runbook job status
- Correlate data from multiple Automation jobs

To run queries, select **Logs** from the **Monitoring** section of your automation account main blade.

The Azure portal provides a few query templates so you can get started. As we can see below, we used an existing Kusto query template to list all completed jobs in the automation account.

The screenshot shows the Azure Log Analytics 'New Query' interface. On the left, the 'Automation Jobs' alert is selected under the 'Alerts' section. The main pane displays a Kusto query:

```

1 // Azure Automation jobs that are Completed
2 // List all automation jobs that got completed.
3 // To create an alert for this query, click '+ New alert rule'
4 AzureDiagnostics
5 | where ResourceProvider == "MICROSOFT.AUTOMATION" and Category == "JobLogs" and ResultType == "Completed"
6 | project TimeGenerated, RunbookName_s, ResultType

```

The results table shows the following data:

TimeGenerated [UTC]	RunbookName_s	ResultType
4/26/2022, 7:24:57.477 P...	DemoDP300	Completed
TimeGenerated [UTC]	2022-04-26T19:24:57.477Z	
RunbookName_s	DemoDP300	
ResultType	Completed	

The ability to forward Azure Automation diagnostic logs to a Log Analytics workspace is an important feature that helps you monitor the health of your runbooks.

Note

Before you start using Log Analytics to query Automation jobs data, you must configure **Diagnostic settings** for your Automation Account.

Monitor elastic jobs

With elastic jobs, job executions are logged, and any failures are automatically retried. The automatic retry feature provides services more resilient to transient failures.

You can monitor elastic jobs execution through Azure portal, PowerShell, and T-SQL.

Azure portal

To view the history of jobs execution, select **Overview** for your Elastic Job agent main blade.

Resource group (change) ContosoRg
 Status Ready
 Location West US 2
 Subscription (change)
 Subscription ID

Server name agentserver.database.windows.net
 Job database JobDatabase

Elastic Jobs is currently in preview and you can see the list of currently executing jobs below. Click here to learn more about how to create and manage your jobs, target groups, and credentials.

Latest 100 job executions

STATUS	JOB NAME	JOB EXECUTION ID	START TIME	END TIME	DURATION
In Progress	Job1	aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb	6/12/2018, 10:48:02 PM		
Succeeded	Job1	bbbbbbb-1111-2222-3333-cccccccccc	6/12/2018, 10:46:21 PM	6/12/2018, 10:46:30 PM	10s
Succeeded	Job1	ccccccc-2222-3333-4444-ddddddddddd	6/12/2018, 10:45:59 PM	6/12/2018, 10:46:16 PM	16s
Succeeded	Job1	ddddddd-3333-4444-5555-eeeeeeeeeee	6/12/2018, 8:29:20 PM	6/12/2018, 8:29:32 PM	12s

PowerShell

The following code snippets get job execution details.

```
# get the latest 5 executions run
$jobAgent | Get-AzSqlElasticJobExecution -Count 5

# get the job step execution details
$jobExecution | Get-AzSqlElasticJobStepExecution

# get the job target execution details
$jobExecution | Get-AzSqlElasticJobTargetExecution -Count 2
```

T-SQL

The following example shows how to view execution status details for all jobs. Create a job agent and connect to the job database, then run the following command:

```
--View all execution status for all jobs
SELECT *
FROM jobs.job_executions
ORDER BY start_time DESC;

--View all execution statuses for job named 'MyJob'
SELECT * FROM jobs.job_executions
WHERE job_name = 'MyJob'
ORDER BY start_time DESC;

-- View all active executions
SELECT *
FROM jobs.job_executions
WHERE is_active = 1
ORDER BY start_time DESC;
```

7. Exercise: Deploy an automation runbook to automatically rebuild indexes

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/7-exercise-deploy-automation-runbook-rebuild-indexes>

Exercise: Deploy an automation runbook to automatically rebuild indexes

Completed

In this exercise, you'll learn how to create an automation runbook to automatically rebuild indexes.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Module assessment

<https://learn.microsoft.com/en-us/training/modules/manage-azure-paas-resources-using-automated-methods/8-knowledge-check>

Module assessment

Completed

9. Summary

Summary

Completed

In this module you learned about various ways to automate common tasks in Azure and within SQL Server. One of the benefits of cloud computing is that a framework for robust automation, monitoring, and tooling is part of the platform.

In the on-premises world, a complex set of tools would need to be assembled just to match the built-in functionality available with Azure metrics, policy, and automation. Additionally, SQL Server provides an automation framework using SQL Agent, and a robust monitoring solution via Extended Events. You can take advantage of both Azure Automation and Azure elastic jobs to automate the management of your Azure resources.