Background jobs

11/05/2018 • 23 minutes to read • Contributors all

In this article

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction--the application can start the job and then continue to process interactive requests from users. This can help to minimize the load on the application UI, which can improve availability and reduce interactive response times.

For example, if an application is required to generate thumbnails of images that are uploaded by users, it can do this as a background job and save the thumbnail to storage when it is complete--without the user needing to wait for the process to be completed. In the same way, a user placing an order can initiate a background workflow that processes the order, while the UI allows the user to continue browsing the web app. When the background job is complete, it can update the stored orders data and send an email to the user that confirms the order.

When you consider whether to implement a task as a background job, the main criteria is whether the task can run without user interaction and without the UI needing to wait for the job to be completed. Tasks that require the user or the UI to wait while they are completed might not be appropriate as background jobs.

Types of background jobs

Background jobs typically include one or more of the following types of jobs:

- CPU-intensive jobs, such as mathematical calculations or structural model analysis.
- I/O-intensive jobs, such as executing a series of storage transactions or indexing files.
- Batch jobs, such as nightly data updates or scheduled processing.
- Long-running workflows, such as order fulfillment, or provisioning services and systems.
- Sensitive-data processing where the task is handed off to a more secure location for processing. For example, you might not want to process sensitive data within a web app. Instead, you might use a pattern such as the Gatekeeper pattern to transfer the data to an isolated background process that has access to protected storage.

Triggers

Background jobs can be initiated in several different ways. They fall into one of the following categories:

- **Event-driven triggers**. The task is started in response to an event, typically an action taken by a user or a step in a workflow.

- Schedule-driven triggers. The task is invoked on a schedule based on a timer. This might be a recurring schedule or a one-off invocation that is specified for a later time.

Event-driven triggers

Event-driven invocation uses a trigger to start the background task. Examples of using event-driven triggers include:

- The UI or another job places a message in a queue. The message contains data about an action that has taken place, such as the user placing an order. The background task listens on this queue and detects the arrival of a new message. It reads the message and uses the data in it as the input to the background job.
- The UI or another job saves or updates a value in storage. The background task monitors the storage and detects changes. It reads the data and uses it as the input to the background job.
- The UI or another job makes a request to an endpoint, such as an HTTPS URI, or an API that is exposed as a web service. It passes the data that is required to complete the background task as part of the request. The endpoint or web service invokes the background task, which uses the data as its input.

Typical examples of tasks that are suited to event-driven invocation include image processing, workflows, sending information to remote services, sending email messages, and provisioning new users in multitenant applications.

Schedule-driven triggers

Schedule-driven invocation uses a timer to start the background task. Examples of using schedule-driven triggers include:

- A timer that is running locally within the application or as part of the application's operating system invokes a background task on a regular basis.
- A timer that is running in a different application, or a timer service such as Azure Scheduler, sends a request to an API or web service on a regular basis. The API or web service invokes the background task.
- A separate process or application starts a timer that causes the background task to be invoked once after a specified time delay, or at a specific time.

Typical examples of tasks that are suited to schedule-driven invocation include batch-processing routines (such as updating related-products lists for users based on their recent behavior), routine data processing tasks (such as updating indexes or generating accumulated results), data analysis for daily reports, data retention cleanup, and data consistency checks.

If you use a schedule-driven task that must run as a single instance, be aware of the following:

- If the compute instance that is running the scheduler (such as a virtual machine using Windows scheduled tasks) is scaled, you will have multiple instances of the scheduler running. These could start multiple instances of the task.
- If tasks run for longer than the period between scheduler events, the scheduler may start another instance of the task while the previous one is still running.

Returning results

Background jobs execute asynchronously in a separate process, or even in a separate location, from the UI or the process that invoked the background task. Ideally, background tasks are "fire and forget" operations, and their execution progress has no impact on the UI or the calling process. This means that the calling process does not wait for completion of the tasks. Therefore, it cannot automatically detect when the task ends.

If you require a background task to communicate with the calling task to indicate progress or completion, you must implement a mechanism for this. Some examples are:

- Write a status indicator value to storage that is accessible to the UI or caller task, which can monitor or check this value when required. Other data that the background task must return to the caller can be placed into the same storage.
- Establish a reply queue that the UI or caller listens on. The background task can send messages to the queue that indicate status and completion. Data that the background task must return to the caller can be placed into the messages. If you are using Azure Service Bus, you can use the **ReplyTo** and **CorrelationId** properties to implement this capability.
- Expose an API or endpoint from the background task that the UI or caller can access to obtain status information. Data that the background task must return to the caller can be included in the response.
- Have the background task call back to the UI or caller through an API to indicate status at predefined points or on completion. This might be through events raised locally or through a publish-and-subscribe mechanism. Data that the background task must return to the caller can be included in the request or event payload.

Hosting environment

You can host background tasks by using a range of different Azure platform services:

- Azure Web Apps and WebJobs. You can use WebJobs to execute custom jobs based on a range of different types of scripts or executable programs within the context of a web app.
- Azure Virtual Machines. If you have a Windows service or want to use the Windows Task Scheduler, it is common to host your background tasks within a dedicated virtual machine.
- Azure Batch. Batch is a platform service that schedules compute-intensive work to run on a managed collection of virtual machines. It can automatically scale compute resources.
- Azure Kubernetes Service (AKS). Azure Kubernetes Service provides a managed hosting environment for Kubernetes on Azure.

The following sections describe each of these options in more detail, and include considerations to help you choose the appropriate option.

Azure Web Apps and WebJobs

You can use Azure WebJobs to execute custom jobs as background tasks within an Azure Web App. WebJobs run within the context of your web app as a continuous process. WebJobs also run in response to a trigger event from Azure Scheduler or external factors, such as changes to storage blobs and message queues. Jobs can be started and stopped on demand, and shut down gracefully. If a continuously running WebJob fails, it is automatically restarted. Retry and error actions are configurable.

When you configure a WebJob:

- If you want the job to respond to an event-driven trigger, you should configure it as **Run continuously**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/continuous.
- If you want the job to respond to a schedule-driven trigger, you should configure it as **Run on a schedule**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/triggered.
- If you choose the **Run on demand** option when you configure a job, it will execute the same code as the **Run on a schedule** option when you start it.

Azure WebJobs run within the sandbox of the web app. This means that they can access environment variables and share information, such as connection strings, with the web app. The job has access to the unique identifier of the machine that is running the job. The connection string named **AzureWebJobsStorage** provides access to Azure storage queues, blobs, and tables for application data, and access to Service Bus for messaging and communication. The connection string named **AzureWebJobsDashboard** provides access to the job action log files.

Azure WebJobs have the following characteristics:

- **Security**: WebJobs are protected by the deployment credentials of the web app.

- **Supported file types**: You can define WebJobs by using command scripts (.cmd), batch files (.bat), PowerShell scripts (.ps1), bash shell scripts (.sh), PHP scripts (.php), Python scripts (.py), JavaScript code (.js), and executable programs (.exe, .jar, and more).
- **Deployment**: You can deploy scripts and executables by using the Azure portal, by using Visual Studio, by using the Azure WebJobs SDK, or by copying them directly to the following locations:
  - For triggered execution: site/wwwroot/app_data/jobs/triggered/{job name}
  - For continuous execution: site/wwwroot/app_data/jobs/continuous/{job name}
- **Logging**: Console.Out is treated (marked) as INFO. Console.Error is treated as ERROR. You can access monitoring and diagnostics information by using the Azure portal. You can download log files directly from the site. They are saved in the following locations:
  - For triggered execution: Vfs/data/jobs/triggered/jobName
  - For continuous execution: Vfs/data/jobs/continuous/jobName
- **Configuration**: You can configure WebJobs by using the portal, the REST API, and PowerShell. You can use a configuration file named settings.job in the same root directory as the job script to provide configuration information for a job. For example:
  - { "stopping_wait_time": 60 }
  - { "is_singleton": true }

Considerations

- By default, WebJobs scale with the web app. However, you can configure jobs to run on single instance by setting the **is_singleton** configuration property to **true**. Single instance WebJobs are useful for tasks that you do not want to scale or run as simultaneous multiple instances, such as reindexing, data analysis, and similar tasks.
- To minimize the impact of jobs on the performance of the web app, consider creating an empty Azure Web App instance in a new App Service plan to host long-running or resource-intensive WebJobs.

Azure Virtual Machines

Background tasks might be implemented in a way that prevents them from being deployed to Azure Web Apps, or these options might not be convenient. Typical examples are Windows services, and third-party utilities and executable programs. Another example might be programs written for an execution environment that is different than that hosting the application. For example, it might be a Unix or Linux program that you want to execute from a Windows or .NET application. You can choose from a range of operating systems for an Azure virtual machine, and run your service or executable on that virtual machine.

To help you choose when to use Virtual Machines, see Azure App Services, Cloud Services and Virtual Machines comparison. For information about the options for Virtual Machines, see Sizes for Windows virtual machines in Azure. For more information about the operating systems and prebuilt images that are available for Virtual Machines, see Azure Virtual Machines Marketplace.

To initiate the background task in a separate virtual machine, you have a range of options:

- You can execute the task on demand directly from your application by sending a request to an endpoint that the task exposes. This passes in any data that the task requires. This endpoint invokes the task.
- You can configure the task to run on a schedule by using a scheduler or timer that is available in your chosen operating system. For example, on Windows you can use Windows Task Scheduler to execute scripts and tasks. Or, if you have SQL Server installed on the virtual machine, you can use the SQL Server Agent to execute scripts and tasks.
- You can use Azure Scheduler to initiate the task by adding a message to a queue that the task listens on, or by sending a request to an API that the task exposes.

See the earlier section Triggers for more information about how you can initiate background tasks.

Considerations

Consider the following points when you are deciding whether to deploy background tasks in an Azure virtual machine:

- Hosting background tasks in a separate Azure virtual machine provides flexibility and allows precise control over initiation, execution, scheduling, and resource allocation. However, it will increase runtime cost if a virtual machine must be deployed just to run background tasks.
- There is no facility to monitor the tasks in the Azure portal and no automated restart capability for failed tasks--although you can monitor the basic status of the virtual machine and manage it by using the Azure Resource Manager Cmdlets. However, there are no facilities to control processes and threads in compute nodes. Typically, using a virtual machine will require additional effort to implement a mechanism that collects data from instrumentation in the task, and from the operating system in the virtual machine. One solution that might be appropriate is to use the System Center Management Pack for Azure.
- You might consider creating monitoring probes that are exposed through HTTP endpoints. The code for these probes could perform health checks, collect operational information and statistics--or collate error information and return it to a management application. For more information, see the Health Endpoint Monitoring pattern.

For more information, see:

- Virtual Machines
- Azure Virtual Machines FAQ

Azure Batch

Consider Azure Batch if you need to run large, parallel high-performance computing (HPC) workloads across tens, hundreds, or thousands of VMs.

The Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling. Azure Batch supports both Linux and Windows VMs.

Considerations

Batch works well with intrinsically parallel workloads. It can also perform parallel calculations with a reduce step at the end, or run Message Passing Interface (MPI) applications for parallel tasks that require message passing between nodes.

An Azure Batch job runs on a pool of nodes (VMs). One approach is to allocate a pool only when needed and then delete it after the job completes. This maximizes utilization, because nodes are not idle, but the job must wait for nodes to be allocated. Alternatively, you can create a pool ahead of time. That approach minimizes the time that it takes for a job to start, but can result in having nodes that sit idle. For more information, see Pool and compute node lifetime.

For more information, see:

- What is Azure Batch?
- Develop large-scale parallel compute solutions with Batch
- Batch and HPC solutions for large-scale computing workloads

Azure Kubernetes Service

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, which makes it easy to deploy and manage containerized applications.

Containers can be useful for running background jobs. Some of the benefits include:

- Containers support high-density hosting. You can isolate a background task in a container, while placing multiple containers in each VM.

- The container orchestrator handles internal load balancing, configuring the internal network, and other configuration tasks.
- Containers can be started and stopped as needed.
- Azure Container Registry allows you to register your containers inside Azure boundaries. This comes with security, privacy, and proximity benefits.

Considerations

- Requires an understanding of how to use a container orchestrator. Depending on the skillset of your DevOps team, this may or may not be an issue.

For more information, see:

- Overview of containers in Azure
- Introduction to private Docker container registries

Partitioning

If you decide to include background tasks within an existing compute instance, you must consider how this will affect the quality attributes of the compute instance and the background task itself. These factors will help you to decide whether to colocate the tasks with the existing compute instance or separate them out into a separate compute instance:

- **Availability**: Background tasks might not need to have the same level of availability as other parts of the application, in particular the UI and other parts that are directly involved in user interaction. Background tasks might be more tolerant of latency, retried connection failures, and other factors that affect availability because the operations can be queued. However, there must be sufficient capacity to prevent the backup of requests that could block queues and affect the application as a whole.

- **Scalability**: Background tasks are likely to have a different scalability requirement than the UI and the interactive parts of the application. Scaling the UI might be necessary to meet peaks in demand, while outstanding background tasks might be completed during less busy times by fewer compute instances.

- **Resiliency**: Failure of a compute instance that just hosts background tasks might not fatally affect the application as a whole if the requests for these tasks can be queued or postponed until the task is available again. If the compute instance and/or tasks can be restarted within an appropriate interval, users of the application might not be affected.

- **Security**: Background tasks might have different security requirements or restrictions than the UI or other parts of the application. By using a separate compute instance, you can specify a different security environment for the tasks. You can also use patterns such as Gatekeeper to isolate the background compute instances from the UI in order to maximize security and separation.

- **Performance**: You can choose the type of compute instance for background tasks to specifically match the performance requirements of the tasks. This might mean using a less expensive compute option if the tasks do not require the same processing capabilities as the UI, or a larger instance if they require additional capacity and resources.

- **Manageability**: Background tasks might have a different development and deployment rhythm from the main application code or the UI. Deploying them to a separate compute instance can simplify updates and versioning.

- **Cost**: Adding compute instances to execute background tasks increases hosting costs. You should carefully consider the trade-off between additional capacity and these extra costs.

For more information, see the Leader Election pattern and the Competing Consumers pattern.

Conflicts

If you have multiple instances of a background job, it is possible that they will compete for access to resources and services, such as databases and storage. This concurrent access can result in resource contention, which might cause conflicts in availability of the services and in the integrity of data in storage. You can resolve resource contention by using a pessimistic locking approach. This prevents competing instances of a task from concurrently accessing a service or corrupting data.

Another approach to resolve conflicts is to define background tasks as a singleton, so that there is only ever one instance running. However, this eliminates the reliability and performance benefits that a multiple-instance configuration can provide. This is especially true if the UI can supply sufficient work to keep more than one background task busy.

It is vital to ensure that the background task can automatically restart and that it has sufficient capacity to cope with peaks in demand. You can achieve this by allocating a compute instance with sufficient resources, by implementing a queueing mechanism that can store requests for later execution when demand decreases, or by using a combination of these techniques.

Coordination

The background tasks might be complex and might require multiple individual tasks to execute to produce a result or to fulfill all the requirements. It is common in these scenarios to divide the task into smaller discreet steps or subtasks that can be executed by multiple consumers. Multistep jobs can be more efficient and more flexible because individual steps might be reusable in multiple jobs. It is also easy to add, remove, or modify the order of the steps.

Coordinating multiple tasks and steps can be challenging, but there are three common patterns that you can use to guide your implementation of a solution:

- **Decomposing a task into multiple reusable steps**. An application might be required to perform a variety of tasks of varying complexity on the information that it processes. A straightforward but inflexible approach to implementing this application might be to perform this processing as a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere within the application. For more information, see the Pipes and Filters pattern.

- **Managing execution of the steps for a task**. An application might perform tasks that comprise a number of steps (some of which might invoke remote services or access remote resources). The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task. For more information, see Scheduler Agent Supervisor pattern.

- **Managing recovery for task steps that fail**. An application might need to undo the work that is performed by a series of steps (which together define an eventually consistent operation) if one or more of the steps fail. For more information, see the Compensating Transaction pattern.

Resiliency considerations

Background tasks must be resilient in order to provide reliable services to the application. When you are planning and designing background tasks, consider the following points:

- Background tasks must be able to gracefully handle restarts without corrupting data or introducing inconsistency into the application. For long-running or multistep tasks, consider using *check pointing* by saving the state of jobs in persistent storage, or as messages in a queue if this is appropriate. For example, you can persist state information in a message in a queue and incrementally update this state information with the task progress so that the task can be processed from the last known good checkpoint--instead of restarting from the beginning.

When using Azure Service Bus queues, you can use message sessions to enable the same scenario. Sessions allow you to save and retrieve the application processing state by using the SetState and GetState methods. For more information about designing reliable multistep processes and workflows, see the Scheduler Agent Supervisor pattern.

- When you use queues to communicate with background tasks, the queues can act as a buffer to store requests that are sent to the tasks while the application is under higher than usual load. This allows the tasks to catch up with the UI during less busy periods. It also means that restarts will not block the UI. For more information, see the Queue-Based Load Leveling pattern. If some tasks are more important than others, consider implementing the Priority Queue pattern to ensure that these tasks run before less important ones.
- Background tasks that are initiated by messages or process messages must be designed to handle inconsistencies, such as messages arriving out of order, messages that repeatedly cause an error (often referred to as *poison messages*), and messages that are delivered more than once. Consider the following:
    - Messages that must be processed in a specific order, such as those that change data based on the existing data value (for example, adding a value to an existing value), might not arrive in the original order in which they were sent. Alternatively, they might be handled by different instances of a background task in a different order due to varying loads on each instance. Messages that must be processed in a specific order should include a sequence number, key, or some other indicator that background tasks can use to ensure that they are processed in the correct order. If you are using Azure Service Bus, you can use message sessions to guarantee the order of delivery. However, it is usually more efficient, where possible, to design the process so that the message order is not important.
    - Typically, a background task will peek at messages in the queue, which temporarily hides them from other message consumers. Then it deletes the messages after they have been successfully processed. If a background task fails when processing a message, that message will reappear on the queue after the peek time-out expires. It will be processed by another instance of the task or during the next processing cycle of this instance. If the message consistently causes an error in the consumer, it will block the task, the queue, and eventually the application itself when the queue becomes full. Therefore, it is vital to detect and remove poison messages from the queue. If you are using Azure Service Bus, messages that cause an error can be moved automatically or manually to an associated dead letter queue.
    - Queues are guaranteed at *least once* delivery mechanisms, but they might deliver the same message more than once. In addition, if a background task fails after processing a message but before deleting it from the queue, the message will become available for processing again. Background tasks should be idempotent, which means that processing the same message more than once does not cause an error or inconsistency in the application's data. Some operations are naturally idempotent, such as setting a stored value to a specific new value. However, operations such as adding a value to an existing stored value without checking that the stored value is still the same as when the message was originally sent will cause inconsistencies. Azure Service Bus queues can be configured to automatically remove duplicated messages.
    - Some messaging systems, such as Azure storage queues and Azure Service Bus queues, support a de-queue count property that indicates the number of times a message has been read from the queue. This can be useful in handling repeated and poison messages. For more information, see Asynchronous Messaging Primer and Idempotency Patterns.

Scaling and performance considerations

Background tasks must offer sufficient performance to ensure they do not block the application, or cause inconsistencies due to delayed operation when the system is under load. Typically, performance is improved by scaling the compute instances that host the background tasks. When you are planning and designing background tasks, consider the following points around scalability and performance:

- Azure supports autoscaling (both scaling out and scaling back in) based on current demand and load or on a predefined schedule, for Web Apps and Virtual Machines hosted deployments. Use this feature to ensure that the application as a whole has sufficient performance capabilities while minimizing runtime costs.
- Where background tasks have a different performance capability from the other parts of an application (for example, the UI or components such as the data access layer), hosting the background tasks together in a separate compute service allows the UI and background tasks to scale independently to manage the load. If multiple background tasks have significantly different performance capabilities from each other, consider dividing them and scaling each type independently. However, note that this might increase runtime costs.
- Simply scaling the compute resources might not be sufficient to prevent loss of performance under load. You might also need to scale storage queues and other resources to prevent a single point of the overall processing chain from becoming a bottleneck. Also, consider other limitations, such as the maximum throughput of storage and other services that the application and the background tasks rely on.
- Background tasks must be designed for scaling. For example, they must be able to dynamically detect the number of storage queues in use in order to listen on or send messages to the appropriate queue.
- By default, WebJobs scale with their associated Azure Web Apps instance. However, if you want a WebJob to run as only a single instance, you can create a Settings.job file that contains the JSON data **{ "is_singleton": true }**. This forces Azure to only run one instance of the WebJob, even if there are multiple instances of the associated web app. This can be a useful technique for scheduled jobs that must run as only a single instance.

Related patterns

- Compute Partitioning Guidance