Optimize the performance and reliability of Azure Functions

10/16/2017 • 3 minutes to read • Contributors  all

In this article

This article provides guidance to improve the performance and reliability of your serverless function apps.

Avoid long running functions

Large, long-running functions can cause unexpected timeout issues. A function can become large due to many Node.js dependencies. Importing dependencies can also cause increased load times that result in unexpected timeouts. Dependencies are loaded both explicitly and implicitly. A single module loaded by your code may load its own additional modules.

Whenever possible, refactor large functions into smaller function sets that work together and return responses fast. For example, a webhook or HTTP trigger function might require an acknowledgment response within a certain time limit; it is common for webhooks to require an immediate response. You can pass the HTTP trigger payload into a queue to be processed by a queue trigger function. This approach allows you to defer the actual work and return an immediate response.

Cross function communication

When integrating multiple functions, it is generally a best practice to use storage queues for cross function communication. The main reason is storage queues are cheaper and much easier to provision.

Individual messages in a storage queue are limited in size to 64 KB. If you need to pass larger messages between functions, an Azure Service Bus queue could be used to support message sizes up to 256 KB.

Service Bus topics are useful if you require message filtering before processing.

Event hubs are useful to support high volume communications.

Write functions to be stateless

Functions should be stateless and idempotent if possible. Associate any required state information with your data. For example, an order being processed would likely have an associated `state` member. A function could process an order based on that state while the function itself remains stateless.

Idempotent functions are especially recommended with timer triggers. For example, if you have something that absolutely must run once a day, write it so it can run any time during the day with the same results. The function can exit when there is no work for a particular day. Also if a previous run failed to complete, the next run should pick up where it left off.

Write defensive functions

Assume your function could encounter an exception at any time. Design your functions with the ability to continue from a previous fail point during the next execution. Consider a scenario that requires the following actions:

1. Query for 10,000 rows in a db.
2. Create a queue message for each of those rows to process further down the line.

Depending on how complex your system is, you may have: involved downstream services behaving badly, networking outages, or quota limits reached, etc. All of these can affect your function at any time. You need to design your functions to be prepared for it.

How does your code react if a failure occurs after inserting 5,000 of those items into a queue for processing? Track items in a set that you've completed. Otherwise, you might insert them again next time. This can have a serious impact on your work flow.

If a queue item was already processed, allow your function to be a no-op.

Take advantage of defensive measures already provided for components you use in the Azure Functions platform. For example, see **Handling poison queue messages** in the documentation for Azure Storage Queue triggers.

Don't mix test and production code in the same function app

Functions within a function app share resources. For example, memory is shared. If you're using a function app in production, don't add test-related functions and resources to it. It can cause unexpected overhead during production code execution.

Be careful what you load in your production function apps. Memory is averaged across each function in the app.

If you have a shared assembly referenced in multiple .Net functions, put it in a common shared folder. Reference the assembly with a statement similar to the following example:

Copy

```
#r "..\Shared\MyAssembly.dll".
```

Otherwise, it is easy to accidentally deploy multiple test versions of the same binary that behave differently between functions.

Don't use verbose logging in production code. It has a negative performance impact.

Use async code but avoid blocking calls

Asynchronous programming is a recommended best practice. However, always avoid referencing the `Result` property or calling `Wait` method on a `Task` instance. This approach can lead to thread exhaustion.

> Tip
>
> If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, review the article Improper Instantiation antipattern.

Next steps

For more information, see the following resources:

Because Azure Functions uses Azure App Service, you should also be aware of App Service guidelines.

- Patterns and Practices HTTP Performance Optimizations