Health checks in ASP.NET Core

Article • 01/01/2024

By Glenn Condron and Juergen Gutsch

ASP.NET Core offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for various real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Health checks are typically used with an external monitoring service or container orchestrator to check the status of an app. Before adding health checks to an app, decide on which monitoring system to use. The monitoring system dictates what types of health checks to create and how to configure their endpoints.

Basic health probe

For many apps, a basic health probe configuration that reports the app's availability to process requests (*liveness*) is sufficient to discover the status of the app.

The basic configuration registers health check services and calls the Health Checks Middleware to respond at a URL endpoint with a health response. By default, no specific health checks are registered to test any particular dependency or subsystem. The app is considered healthy if it can respond at the health endpoint URL. The default response writer writes HealthStatus as a plaintext response to the client. The `HealthStatus` is HealthStatus.Healthy, HealthStatus.Degraded, or HealthStatus.Unhealthy.

Register health check services with AddHealthChecks in `Program.cs`. Create a health check endpoint by calling MapHealthChecks.

The following example creates a health check endpoint at `/healthz`:

C#

```csharp
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHealthChecks();

var app = builder.Build();

app.MapHealthChecks("/healthz");

app.Run();
```

Docker `HEALTHCHECK`

Docker offers a built-in `HEALTHCHECK` directive that can be used to check the status of an app that uses the basic health check configuration:

Dockerfile

```dockerfile
HEALTHCHECK CMD curl --fail http://localhost:5000/healthz || exit
```

The preceding example uses `curl` to make an HTTP request to the health check endpoint at `/healthz`. `curl` isn't included in the .NET Linux container images, but it can be added by installing the required package in the Dockerfile. Containers that use images based on Alpine Linux can use the included `wget` in place of `curl`.

Create health checks

Health checks are created by implementing the IHealthCheck interface. The CheckHealthAsync method returns a HealthCheckResult that indicates the health as `Healthy`, `Degraded`, or `Unhealthy`. The result is written as a plaintext response with a configurable status code. Configuration is described in the Health check options section. HealthCheckResult can also return optional key-value pairs.

The following example demonstrates the layout of a health check:

C#

```csharp
public class SampleHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken = default)
    {
        var isHealthy = true;

        // ...

        if (isHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result."));
        }

        return Task.FromResult(
            new HealthCheckResult(
                context.Registration.FailureStatus, "An unhealthy result."));
    }
}
```

The health check's logic is placed in the CheckHealthAsync method. The preceding example sets a dummy variable, `isHealthy`, to `true`. If the value of `isHealthy` is set to `false`, the HealthCheckRegistration.FailureStatus status is returned.

If CheckHealthAsync throws an exception during the check, a new HealthReportEntry is returned with its HealthReportEntry.Status set to the FailureStatus. This status is defined by AddCheck (see the Register health check services section) and includes the inner exception that caused the check failure. The Description is set to the exception's message.

Register health check services

To register a health check service, call AddCheck in `Program.cs`:

C#

```csharp
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>("Sample");
```

The AddCheck overload shown in the following example sets the failure status (HealthStatus) to report when the health check reports a failure. If the failure status is set to `null` (default), HealthStatus.Unhealthy is reported. This overload is a useful scenario for library authors, where the failure status indicated by the library is enforced by the app when a health check failure occurs if the health check implementation honors the setting.

Tags can be used to filter health checks. Tags are described in the Filter health checks section.

C#

```csharp
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>(
        "Sample",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "sample" });
```

AddCheck can also execute a lambda function. In the following example, the health check always returns a healthy result:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck("Sample", () => HealthCheckResult.Healthy("A healthy result."));
```

Call AddTypeActivatedCheck to pass arguments to a health check implementation. In the following example, a type-activated health check accepts an integer and a string in its constructor:

C#

```
public class SampleHealthCheckWithArgs : IHealthCheck
{
    private readonly int _arg1;
    private readonly string _arg2;

    public SampleHealthCheckWithArgs(int arg1, string arg2)
        => (_arg1, _arg2) = (arg1, arg2);

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken = default)
    {
        // ...

        return Task.FromResult(HealthCheckResult.Healthy("A healthy result."));
    }
}
```

To register the preceding health check, call `AddTypeActivatedCheck` with the integer and string passed as arguments:

C#

```
builder.Services.AddHealthChecks()
    .AddTypeActivatedCheck<SampleHealthCheckWithArgs>(
        "Sample",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "sample" },
        args: new object[] { 1, "Arg" });
```

Use Health Checks Routing

In `Program.cs`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

C#

```
app.MapHealthChecks("/healthz");
```

Require host

Call `RequireHost` to specify one or more permitted hosts for the health check endpoint. Hosts should be Unicode rather than punycode and may include a port. If a collection isn't supplied, any host is accepted:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("www.contoso.com:5001");
```

To restrict the health check endpoint to respond only on a specific port, specify a port in the call to `RequireHost`. This approach is typically used in a container environment to expose a port for monitoring services:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("*:5001");
```

⚠ **Warning**

API that relies on the Host header , such as HttpRequest.Host and RequireHost, are subject to potential spoofing by clients.

To prevent host and port spoofing, use one of the following approaches:

- Use HttpContext.Connection (ConnectionInfo.LocalPort) where the ports are checked.
- Employ Host filtering.

To prevent unauthorized clients from spoofing the port, call RequireAuthorization:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("*:5001")
    .RequireAuthorization();
```

For more information, see Host matching in routes with RequireHost.

Require authorization

Call RequireAuthorization to run Authorization Middleware on the health check request endpoint. A `RequireAuthorization` overload accepts one or more authorization policies. If a policy isn't provided, the default authorization policy is used:

C#

```
app.MapHealthChecks("/healthz")
    .RequireAuthorization();
```

Enable Cross-Origin Requests (CORS)

Although running health checks manually from a browser isn't a common scenario, CORS Middleware can be enabled by calling RequireCors on the health checks endpoints. The RequireCors overload accepts a CORS policy builder delegate (`CorsPolicyBuilder`) or a policy name. For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core.

Health check options

HealthCheckOptions provide an opportunity to customize health check behavior:

- Filter health checks
- Customize the HTTP status code
- Suppress cache headers
- Customize output

Filter health checks

By default, the Health Checks Middleware runs all registered health checks. To run a subset of health checks, provide a function that returns a boolean to the Predicate option.

The following example filters the health checks so that only those tagged with `sample` run:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("sample")
});
```

Customize the HTTP status code

Use ResultStatusCodes to customize the mapping of health status to HTTP status codes. The following StatusCodes assignments are the default values used by the middleware. Change the status code values to meet your requirements:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResultStatusCodes =
    {
        [HealthStatus.Healthy] = StatusCodes.Status200OK,
        [HealthStatus.Degraded] = StatusCodes.Status200OK,
        [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
    }
});
```

Suppress cache headers

AllowCachingResponses controls whether the Health Checks Middleware adds HTTP headers to a probe response to prevent response caching. If the value is `false` (default), the middleware sets or overrides the `Cache-Control`, `Expires`, and `Pragma` headers to prevent response caching. If the value is `true`, the middleware doesn't modify the cache headers of the response:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    AllowCachingResponses = true
});
```

Customize output

To customize the output of a health checks report, set the HealthCheckOptions.ResponseWriter property to a delegate that writes the response:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResponseWriter = WriteResponse
});
```

The default delegate writes a minimal plaintext response with the string value of HealthReport.Status. The following custom delegate outputs a custom JSON response using System.Text.Json:

C#

```
private static Task WriteResponse(HttpContext context, HealthReport healthReport)
{
    context.Response.ContentType = "application/json; charset=utf-8";

    var options = new JsonWriterOptions { Indented = true };

    using var memoryStream = new MemoryStream();
    using (var jsonWriter = new Utf8JsonWriter(memoryStream, options))
    {
        jsonWriter.WriteStartObject();
```

```
            jsonWriter.WriteString("status", healthReport.Status.ToString());
            jsonWriter.WriteStartObject("results");

            foreach (var healthReportEntry in healthReport.Entries)
            {
                jsonWriter.WriteStartObject(healthReportEntry.Key);
                jsonWriter.WriteString("status",
                    healthReportEntry.Value.Status.ToString());
                jsonWriter.WriteString("description",
                    healthReportEntry.Value.Description);
                jsonWriter.WriteStartObject("data");

                foreach (var item in healthReportEntry.Value.Data)
                {
                    jsonWriter.WritePropertyName(item.Key);

                    JsonSerializer.Serialize(jsonWriter, item.Value,
                        item.Value?.GetType() ?? typeof(object));
                }

                jsonWriter.WriteEndObject();
                jsonWriter.WriteEndObject();
            }

            jsonWriter.WriteEndObject();
            jsonWriter.WriteEndObject();
        }

    return context.Response.WriteAsync(
        Encoding.UTF8.GetString(memoryStream.ToArray()));
}
```

The health checks API doesn't provide built-in support for complex JSON return formats because the format is specific to your choice of monitoring system. Customize the response in the preceding examples as needed. For more information on JSON serialization with `System.Text.Json`, see How to serialize and deserialize JSON in .NET.

Database probe

A health check can specify a database query to run as a boolean test to indicate if the database is responding normally.

AspNetCore.Diagnostics.HealthChecks , a health check library for ASP.NET Core apps, includes a health check that runs against a SQL Server database. `AspNetCore.Diagnostics.HealthChecks` executes a `SELECT 1` query against the database to confirm the connection to the database is healthy.

⚠ Warning

When checking a database connection with a query, choose a query that returns quickly. The query approach runs the risk of overloading the database and degrading its performance. In most cases, running a test query isn't necessary. Merely making a successful connection to the database is sufficient. If you find it necessary to run a query, choose a simple SELECT query, such as `SELECT 1`.

To use this SQL Server health check, include a package reference to the AspNetCore.HealthChecks.SqlServer NuGet package. The following example registers the SQL Server health check:

C#

```csharp
builder.Services.AddHealthChecks()
    .AddSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection"));
```

ⓘ Note

AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Entity Framework Core DbContext probe

The `DbContext` check confirms that the app can communicate with the database configured for an EF Core `DbContext`. The `DbContext` check is supported in apps that:

- Use Entity Framework (EF) Core.
- Include a package reference to the Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore NuGet package.

AddDbContextCheck registers a health check for a DbContext. The `DbContext` is supplied to the method as the `TContext`. An overload is available to configure the failure status, tags, and a custom test query.

By default:

- The `DbContextHealthCheck` calls EF Core's `CanConnectAsync` method. You can customize what operation is run when checking health using `AddDbContextCheck` method overloads.
- The name of the health check is the name of the `TContext` type.

The following example registers a `DbContext` and an associated `DbContextHealthCheck`:

C#

```csharp
builder.Services.AddDbContext<SampleDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddHealthChecks()
    .AddDbContextCheck<SampleDbContext>();
```

Separate readiness and liveness probes

In some hosting scenarios, a pair of health checks is used to distinguish two app states:

- *Readiness* indicates if the app is running normally but isn't ready to receive requests.
- *Liveness* indicates if an app has crashed and must be restarted.

Consider the following example: An app must download a large configuration file before it's ready to process requests. We don't want the app to be restarted if the initial download fails because the app can retry downloading the file several times. We use a *liveness probe* to describe the liveness of the process, no other checks are run. We also want to prevent requests from being sent to the app before the configuration file download has succeeded. We use a *readiness probe* to indicate a "not ready" state until the download succeeds and the app is ready to receive requests.

The following background task simulates a startup process that takes roughly 15 seconds. Once it completes, the task sets the `StartupHealthCheck.StartupCompleted` property to true:

C#

```
public class StartupBackgroundService : BackgroundService
{
    private readonly StartupHealthCheck _healthCheck;

    public StartupBackgroundService(StartupHealthCheck healthCheck)
        => _healthCheck = healthCheck;

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // Simulate the effect of a long-running task.
        await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);

        _healthCheck.StartupCompleted = true;
    }
}
```

The `StartupHealthCheck` reports the completion of the long-running startup task and exposes the `StartupCompleted` property that gets set by the background service:

C#

```
public class StartupHealthCheck : IHealthCheck
{
    private volatile bool _isReady;

    public bool StartupCompleted
    {
        get => _isReady;
        set => _isReady = value;
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken = default)
    {
        if (StartupCompleted)
        {
            return Task.FromResult(HealthCheckResult.Healthy("The startup task has completed."));
        }

        return Task.FromResult(HealthCheckResult.Unhealthy("That startup task is still running."));
    }
}
```

The health check is registered with AddCheck in `Program.cs` along with the hosted service. Because the hosted service must set the property on the health check, the health check is also registered in the service container as a singleton:

C#

```
builder.Services.AddHostedService<StartupBackgroundService>();
builder.Services.AddSingleton<StartupHealthCheck>();
builder.Services.AddHealthChecks()
    .AddCheck<StartupHealthCheck>(
        "Startup",
        tags: new[] { "ready" });
```

To create two different health check endpoints, call `MapHealthChecks` twice:

C#

```
app.MapHealthChecks("/healthz/ready", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("ready")
});

app.MapHealthChecks("/healthz/live", new HealthCheckOptions
{
    Predicate = _ => false
});
```

The preceding example creates the following health check endpoints:

- `/healthz/ready` for the readiness check. The readiness check filters health checks to those tagged with `ready`.
- `/healthz/live` for the liveness check. The liveness check filters out all health checks by returning `false` in the HealthCheckOptions.Predicate delegate. For more information on filtering health checks, see Filter health checks in this article.

Before the startup task completes, the `/healthz/ready` endpoint reports an `Unhealthy` status. Once the startup task completes, this endpoint reports a `Healthy` status. The `/healthz/live` endpoint excludes all checks and reports a `Healthy` status for all calls.

Kubernetes example

Using separate readiness and liveness checks is useful in an environment such as Kubernetes. In Kubernetes, an app might be required to run time-consuming startup work before accepting requests, such as a test of the underlying database availability. Using separate checks allows the orchestrator to distinguish whether the app is functioning but not yet ready or if the app has failed to start. For more information on readiness and liveness probes in Kubernetes, see Configure Liveness and Readiness Probes in the Kubernetes documentation.

The following example demonstrates a Kubernetes readiness probe configuration:

YAML

```yaml
spec:
  template:
  spec:
    readinessProbe:
      # an http probe
      httpGet:
        path: /healthz/ready
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
      - containerPort: 80
```

Distribute a health check library

To distribute a health check as a library:

1. Write a health check that implements the IHealthCheck interface as a standalone class. The class can rely on dependency injection (DI), type activation, and named options to access configuration data.

2. Write an extension method with parameters that the consuming app calls in its `Program.cs` method. Consider the following example health check, which accepts `arg1` and `arg2` as constructor parameters:

C#

```csharp
public SampleHealthCheckWithArgs(int arg1, string arg2)
    => (_arg1, _arg2) = (arg1, arg2);
```

The preceding signature indicates that the health check requires custom data to process the health check probe logic. The data is provided to the delegate used to create the health check instance when the health check is registered with an extension method. In the following example, the caller specifies:

- `arg1`: An integer data point for the health check.
- `arg2`: A string argument for the health check.
- `name`: An optional health check name. If `null`, a default value is used.
- `failureStatus`: An optional HealthStatus, which is reported for a failure status. If `null`, HealthStatus.Unhealthy is used.
- `tags`: An optional `IEnumerable<string>` collection of tags.

C#

```csharp
public static class SampleHealthCheckBuilderExtensions
{
    private const string DefaultName = "Sample";

    public static IHealthChecksBuilder AddSampleHealthCheck(
        this IHealthChecksBuilder healthChecksBuilder,
        int arg1,
        string arg2,
        string? name = null,
        HealthStatus? failureStatus = null,
        IEnumerable<string>? tags = default)
    {
        return healthChecksBuilder.Add(
            new HealthCheckRegistration(
                name ?? DefaultName,
                _ => new SampleHealthCheckWithArgs(arg1, arg2),
                failureStatus,
                tags));
    }
}
```

Health Check Publisher

When an IHealthCheckPublisher is added to the service container, the health check system periodically executes your health checks and calls PublishAsync with the result. This process is useful in a push-based health monitoring system scenario that expects each process to call the monitoring system periodically to determine health.

HealthCheckPublisherOptions allow you to set the:

- Delay: The initial delay applied after the app starts before executing IHealthCheckPublisher instances. The delay is applied once at startup and doesn't apply to later iterations. The default value is five seconds.
- Period: The period of IHealthCheckPublisher execution. The default value is 30 seconds.
- Predicate: If Predicate is `null` (default), the health check publisher service runs all registered health checks. To run a subset of health checks, provide a function that filters the set of checks. The predicate is evaluated each period.
- Timeout: The timeout for executing the health checks for all IHealthCheckPublisher instances. Use InfiniteTimeSpan to execute without a timeout. The default value is 30 seconds.

The following example demonstrates the layout of a health publisher:

C#

```
public class SampleHealthCheckPublisher : IHealthCheckPublisher
{
    public Task PublishAsync(HealthReport report, CancellationToken cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy)
        {
            // ...
        }
        else
        {
            // ...
        }

        return Task.CompletedTask;
    }
}
```

The HealthCheckPublisherOptions class provides properties for configuring the behavior of the health check publisher.

The following example registers a health check publisher as a singleton and configures HealthCheckPublisherOptions:

C#

```
builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = healthCheck => healthCheck.Tags.Contains("sample");
});

builder.Services.AddSingleton<IHealthCheckPublisher, SampleHealthCheckPublisher>();
```

ⓘ Note

AspNetCore.Diagnostics.HealthChecks includes publishers for several systems, including Application Insights.

AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Dependency Injection and Health Checks

It's possible to use dependency injection to consume an instance of a specific `Type` inside a Health Check class. Dependency injection can be useful to inject options or a global configuration to a Health Check. Using dependency injection is ***not*** a common scenario to configure Health Checks. Usually, each Health Check is quite specific to the actual test and is configured using `IHealthChecksBuilder` extension methods.

The following example shows a sample Health Check that retrieves a configuration object via dependency injection:

C#

```csharp
public class SampleHealthCheckWithDI : IHealthCheck
{
    private readonly SampleHealthCheckWithDiConfig _config;

    public SampleHealthCheckWithDI(SampleHealthCheckWithDiConfig config)
        => _config = config;

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken = default)
    {
        var isHealthy = true;

        // use _config ...

        if (isHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result."));
        }
```

```
            return Task.FromResult(
                new HealthCheckResult(
                    context.Registration.FailureStatus, "An unhealthy result."));
        }
    }
```

The `SampleHealthCheckWithDiConfig` and the Health check needs to be added to the service container :

C#

```
builder.Services.AddSingleton<SampleHealthCheckWithDiConfig>(new SampleHealthCheckWithDiConfig
{
    BaseUriToCheck = new Uri("https://sample.contoso.com/api/")
});
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheckWithDI>(
        "With Dependency Injection",
        tags: new[] { "inject" });
```

UseHealthChecks vs. MapHealthChecks

There are two ways to make health checks accessible to callers:

- UseHealthChecks registers middleware for handling health checks requests in the middleware pipeline.
- MapHealthChecks registers a health checks endpoint. The endpoint is matched and executed along with other endpoints in the app.

The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to use endpoint aware middleware, such as authorization, and to have greater fine-grained control over the matching policy. The primary advantage of using `UseHealthChecks` over `MapHealthChecks` is controlling exactly where health checks runs in the middleware pipeline.

UseHealthChecks:

- Terminates the pipeline when a request matches the health check endpoint. Short-circuiting is often desirable because it avoids unnecessary work, such as logging and other middleware.
- Is primarily used for configuring the health check middleware in the pipeline.
- Can match any path on a port with a `null` or empty `PathString`. Allows performing a health check on any request made to the specified port.
- Source code

MapHealthChecks allows:

- Terminating the pipeline when a request matches the health check endpoint, by calling ShortCircuit. For example, `app.MapHealthChecks("/healthz").ShortCircuit();`. For more information, see Short-circuit middleware after routing.
- Mapping specific routes or endpoints for health checks.
- Customization of the URL or path where the health check endpoint is accessible.
- Mapping multiple health check endpoints with different routes or configurations. Multiple endpoint support:
  - Enables separate endpoints for different types of health checks or components.
  - Is used to differentiate between different aspects of the app's health or apply specific configurations to subsets of health checks.
- Source code

Additional resources

- View or download sample code (how to download)

Note

This article was partially created with the help of artificial intelligence. Before publishing, an author reviewed and revised the content as needed. See **Our principles for using AI-generated content in Microsoft Learn**.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback