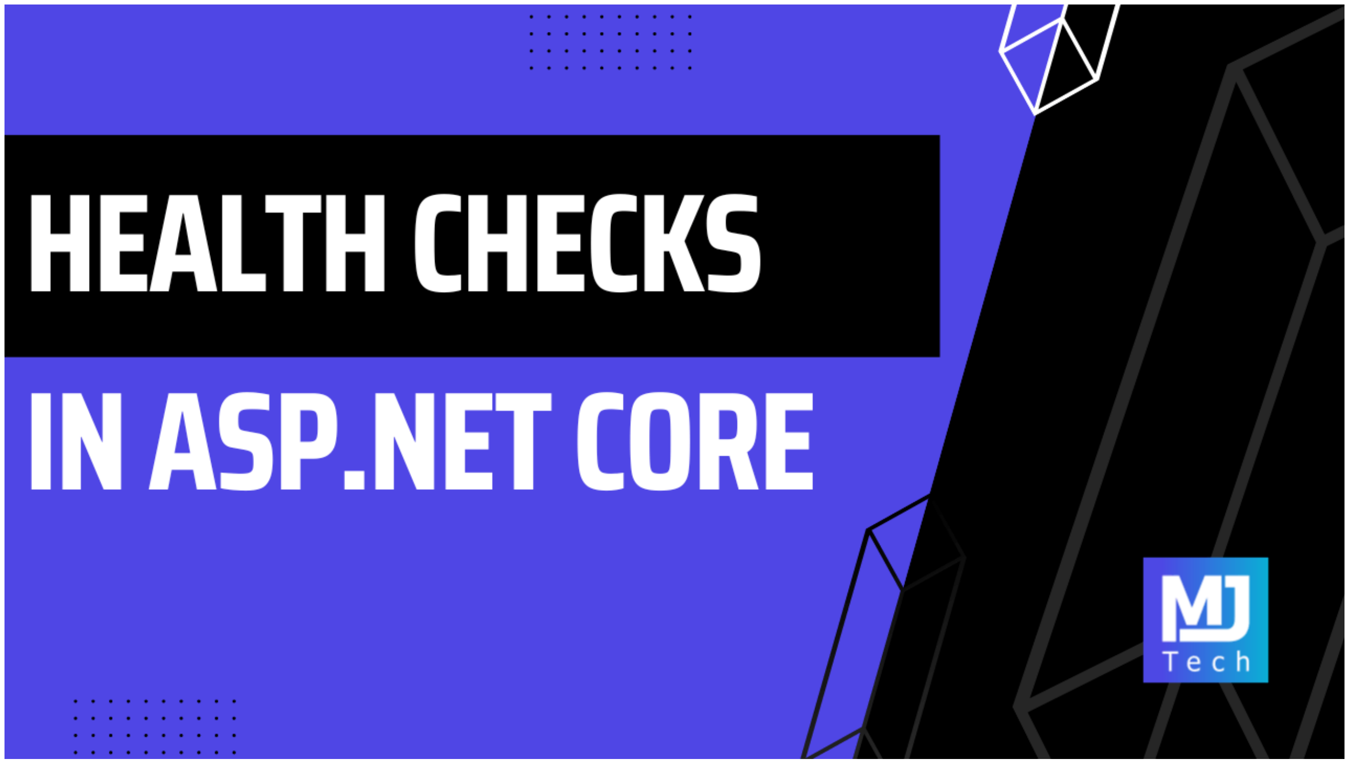

Health Checks In ASP.NET Core For

APRIL 29, 2023

READ TIME - 5 MINUTES

Thank you to our sponsors who keep this newsletter free to the reader:

Today's issue is sponsored by **Rebus Pro.** Rebus is a free .NET "service bus", and **Rebus Pro is the perfect one-up for serious Rebus users.** Use Fleet Manager to get Slack alerts when something fails and retry dead-lettered messages with a click of the mouse.

We all want to build **robust** and **reliable** applications that can scale indefinitely and handle any number of requests.

But with **distributed systems** and **microservices architectures** growing in complexity, it's becoming increasingly harder to **monitor** the **health** of our applications.

It's vital that you have a system in place to receive quick feedback of your application **health.**

That's where **health checks** come in.

- Databases
- APIs
- Caches
- External services

Here's what I'll show you in this week's newsletter:

- What are health checks
- Adding a custom health check
- Using existing health check libraries
- Customizing the health checks response format

Let's see how to implement **health checks** in **ASP.NET Core**.

verifying the **health** and **availability** of an application in **ASP.NET Core.**

ASP.NET Core has **built-in support** for implementing **health checks.**

Here's the basic configuration, which registers the health check services and adds the `HealthCheckMiddleware` to respond at the specified URL.

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

builder.Services.AddHealthChecks();

var app = builder.Build();

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

Share This Article On:

The health check returns a `HealthStatus` value indicating the health of the service.

There are three distinct `HealthStatus` values:

- `HealthStatus.Healthy`
- `HealthStatus.Degraded`
- `HealthStatus.Unhealthy`

You can use the `HealthStatus` to indicate the different states of your application.

For example, if the application is functioning slower than expected you can return `HealthStatus.Degraded`.

IHealthCheck interface.

For example, you can implement a check to see if your **SQL** database is available.

It's important to use a query that can complete quickly in the database, like **SELECT 1**.

Here's a **custom health check** implementation example in the **SqlHealthCheck** class:

```
public class SqlHealthCheck : IHealthCheck
{
    private readonly string _connectionString;

    public SqlHealthCheck(IConfiguration configuration)
    {
        _connectionString = configuration.GetConnectionString("Da
    {
        try
        {
            using var sqlConnection = new SqlConnection(_connecti

            await sqlConnection.OpenAsync(cancellationToken);

            using var command = sqlConnection.CreateCommand();
            command.CommandText = "SELECT 1";

            await command.ExecuteScalarAsync(cancellationToken);

            return HealthCheckResult.Healthy();
        }
        catch(Exception ex)
        {
            return HealthCheckResult.Unhealthy(
                context.Registration.FailureStatus,
                exception: ex);
        }
    }
}
```

The previous call to **`AddHealthChecks`** now becomes:

```
builder.Services.AddHealthChecks()
    .AddCheck<SqlHealthCheck>("custom-sql", HealthStatus.Unhealth
```

We're giving it a custom name and setting which status to use as the failure result in **`HealthCheckContext.Registration.FailureStatus`**.

But stop and think for a moment.

Do you want to implement a **custom health check** on your own for **every external service** that you have?

Of course not! There's a better solution.

everything, you should first see if there's already an **existing library.**

In the `AspNetCore.Diagnostics.HealthChecks` repository you can find a wide collection **health check** packages for frequently used services and libraries.

Here are just a few examples:

- SQL Server - `AspNetCore.HealthChecks.SqlServer`
- Postgres - `AspNetCore.HealthChecks.Npgsql`
- Redis - `AspNetCore.HealthChecks.Redis`
- RabbitMQ - `AspNetCore.HealthChecks.RabbitMQ`
- AWS S3 - `AspNetCore.HealthChecks.Aws.S3`

RabbitMQ:

```
builder.Services.AddHealthChecks()
    .AddCheck<SqlHealthCheck>("custom-sql", HealthStatus.Unhealth
    .AddNpgSql(pgConnectionString)
    .AddRabbitMQ(rabbitConnectionString)
```

Formatting Health Checks Response

By default, the endpoint returning you **health check** status will return a string value representing a `HealthStatus`.

This isn't practical if you have **multiple health checks** configured, as you'd want to view the health status individually entire response will return **`Unhealthy`** and you don't know what's causing the issue.

You can solve this by providing a **`ResponsWriter`**, and there's an existing one in the **`AspNetCore.HealthChecks.UI.Client`** library.

Let's install the **NuGet** package:

```
Install-Package AspNetCore.HealthChecks.UI.Client
```

And you need to slightly update the call to **`MapHealthChecks`** to use the **`ResponseWriter`** coming from this library:

```
app.MapHealthChecks(
    "/health",
```

After making these changes, here's what the response from the health check endpoint looks like:

```
{
  "status": "Unhealthy",
  "totalDuration": "00:00:00.3285211",
  "entries": {
    "npgsql": {
      "data": {},
      "duration": "00:00:00.1183517",
      "status": "Healthy",
      "tags": []
    },
    "rabbitmq": {
      "data": {},
      "duration": "00:00:00.1189561",
      "status": "Healthy",
      "tags": []
    },
      "status": "Unhealthy",
      "tags": []
    }
  }
}
```

Takeaway

Application monitoring is important to track availability, resource usage, and changes to performance in your application.

I've used **health checks** before to implement **failover scenarios** in a **cloud deployment**. When one application

It's easy to monitor the health of your ASP.NET Core applications by **exposing health checks** for your services.

You can decide to implement **custom health checks**, but first consider if there are **existing solutions**.

Thank you for reading, and have an awesome Saturday.

Whenever you're ready, there are 3 ways I can help you:

1. Pragmatic Clean Architecture: This comprehensive course will teach you the system I use to ship production-ready applications using Clean Architecture.

with access to the source code I use in my YouTube videos and exclusive discounts for my courses. Join 970+ engineers here.

3. Promote yourself to 37,000+ subscribers by sponsoring this newsletter.

Become a Better .NET Software Engineer

Join 37,000+ engineers who are improving their skills every Saturday morning.

Milan Jovanović Newsletter Sponsorship Courses Log in