Health checks are crucial for monitoring the status of your applications and services. They provide a quick and automated way to verify that your application’s dependencies and components are running smoothly.

This article will explore how to implement health checks in a .NET 8 web application.

**Why Health Checks in ASP.NET Applications?**

Health checks help you proactively identify issues within your application, allowing you to address them before they impact users. Regularly verifying your application components' health can ensure a more reliable and resilient system.

When developing an ASP.NET application, it often relies on various sub-systems, such as databases, file systems, APIs, and more. One of the most common scenarios involves dependency on a database. Virtually every application requires seamless interaction with a database, making it a critical system component. However, traditional application development has often left this aspect unattended, leading to potential breakdowns if the connection to the database is lost for any reason.

Consider a scenario where your application is dependent on a database. If, for various reasons, the connection to the database is lost, the application is likely to break. While this situation is a fundamental example of why health checks are beneficial, it doesn’t capture the full scope of their development. So,

**Let’s dig deeper into the database example to see why ASP.NET health checks matter on a larger scale.**

* **What if you were able to check whether the database was available before establishing a connection?**— — Imagine having the capability to assess the health and availability of your application’s crucial sub-systems before actively engaging with them. Picture a scenario where, rather than encountering a sudden application failure due to a lost database connection, you could proactively verify its availability.
* **What if your application could gracefully handle a database unavailability scenario?**— — Imagine empowering your application to display a user-friendly message when the database is inaccessible. Instead of a cryptic error message confusing your users, you could seamlessly communicate the unavailability of a critical component, ensuring a better user experience.
* **What if you could seamlessly switch to a fallback database in case of unavailability?**— — Consider the flexibility of instructing your application to seamlessly transition to an alternative database when the primary one becomes unavailable. This not only maintains application continuity but also ensures that users experience minimal disruption.
* **What if you could instruct a load balancer to switch to a fallback environment based on health checks?  
  — —**Picture having the ability to communicate the health status of your application to a load balancer. If the application is deemed unhealthy due to a missing database or any other critical issue, the load balancer can intelligently redirect traffic to a fallback environment, ensuring continuous service availability.

**With ASP.NET Health Checks, you can:**

* Assess the health and availability of your sub-systems.
* Create an endpoint to inform other systems about your application’s health.
* Consume health check endpoints of other systems.

These health checks are specially designed for microservice environments, where loosely connected applications rely on knowing the health of their dependent systems. However, they are also beneficial in monolithic applications that depend on various subsystems and infrastructure.

**How to implement health checks in .Net 8?**

I'll show the health check configuration in two ways in .Net 8

**Section 1: Basic Health Checks Setup**

This section aims to establish a foundation for health checks in your application.

**Required NuGet Packages**Ensure you have the following NuGet package installed:  
1. *Microsoft.Extensions.Diagnostics.HealthChecks*

**Adding Health Checks Services**In your Program.cs, add the required health check services to the Dependency Injection container:  
**Program.cs**

|  |
| --- |
| builder.Services.AddHealthChecks();  //HealthCheck Middleware app.MapHealthChecks("/api/health"); |

After adding these to your .NET 8 web API, run the application successfully. Navigate to the following endpoint using a browser, assuming your application is running at:  
<https://localhost:44333/swagger/feedbackservice/index.html>  
To check the health of your web API, go to:  
https://localhost:44333/api/health

After calling the endpoint, you will see that your web API is “Healthy”.

A screenshot of a computer

Description automatically generated

Upon calling this endpoint, you should observe that your web API is marked as “Healthy.” It’s important to note that, at this stage, we’ve set up basic health checks to ensure the overall health of the application, but specific health checks for subsystems **have not** been implemented yet.

**Section 2: Implemented with Health Checks**

**Custom Health Checks for Enhanced Monitoring**In this section, we’ll dive into specific examples of implementing custom health checks to monitor critical components of your .NET 8 web API. These checks go beyond the basic setup, providing a more granular and insightful view of the health of your application.

**Required NuGet Packages**Ensure you have the following NuGet package installed:  
1. *Microsoft.Extensions.Diagnostics.HealthChecks  
2. AspNetCore.HealthChecks.SqlServer  
3. AspNetCore.HealthChecks.UI  
4. AspNetCore.HealthChecks.UI.Client  
5. AspNetCore.HealthChecks.UI.InMemory.Storage  
6. AspNetCore.HealthChecks.Uris*

**Note:** I have separately created a file called**HealthCheck.cs** , and implemented all the health check configurations.

**a. Database Health Check**

The database health check is a crucial aspect of monitoring the well-being of your application, especially when it relies on a database for storing and retrieving data. This health check ensures that the database is not only reachable but also responsive to queries.

**HealthCheck.cs**

|  |
| --- |
| public static void ConfigureHealthChecks(this IServiceCollection services,IConfiguration configuration) {  services.AddHealthChecks()  .AddSqlServer(configuration["ConnectionStrings:DefaultConnection"], healthQuery: "select 1", name: "SQL Server", failureStatus: HealthStatus.Unhealthy, tags: new[] { "Feedback", "Database" });   //services.AddHealthChecksUI();  services.AddHealthChecksUI(opt =>  {  opt.SetEvaluationTimeInSeconds(10); //time in seconds between check   opt.MaximumHistoryEntriesPerEndpoint(60); //maximum history of checks   opt.SetApiMaxActiveRequests(1); //api requests concurrency   opt.AddHealthCheckEndpoint("feedback api", "/api/health"); //map health check api    })  .AddInMemoryStorage(); } |

**configuration["ConnectionStrings:DefaultConnection"]**This retrieves the connection string from your configuration, allowing flexibility in configuring the database connection.  
**failureStatus: HealthStatus.Unhealthy**This indicates that if the health check fails, the overall health status should be marked as unhealthy.

**Program.cs**Configure the *ConfigureHealthChecks()* inside the *program.cs*

|  |
| --- |
| //Congiguring Health Ckeck builder.Services.ConfigureHealthChecks(builder.Configuration);  //HealthCheck Middleware app.MapHealthChecks("/api/health", new HealthCheckOptions() {  Predicate = \_ => true,  ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse }); app.UseHealthChecksUI(delegate (Options options)  {  options.UIPath = "/healthcheck-ui";  options.AddCustomStylesheet("./HealthCheck/Custom.css");  }); |

**Output:**Endpoint: */api/health*

A screenshot of a computer program

Description automatically generated

Endpoint: */healthcheck-ui*

A screenshot of a computer

Description automatically generated

**b. Remote Endpoints Health Check**

Next, we’ll implement a health check for remote endpoints and memory.  
**RemoteHealthCheck.cs**

|  |
| --- |
| using Microsoft.Extensions.Diagnostics.HealthChecks; using System; using System.Net.Http; using System.Threading; using System.Threading.Tasks;  namespace FeedbackService.Api {  public class RemoteHealthCheck : IHealthCheck  {  private readonly IHttpClientFactory \_httpClientFactory;  public RemoteHealthCheck(IHttpClientFactory httpClientFactory)  {  \_httpClientFactory = httpClientFactory;  }  public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context, CancellationToken cancellationToken = new CancellationToken())  {  using (var httpClient = \_httpClientFactory.CreateClient())  {  var response = await httpClient.GetAsync("https://api.ipify.org");  if (response.IsSuccessStatusCode)  {  return HealthCheckResult.Healthy($"Remote endpoints is healthy.");  }   return HealthCheckResult.Unhealthy("Remote endpoint is unhealthy");  }  }  } } |

This health check monitors the health of a remote endpoint (e.g., an API) by making an HTTP request.

**b. Memory Health Check**

Finally, let’s implement a health check to monitor the memory status of the API service.

**MemoryHealthCheck.cs**

|  |
| --- |
| using System; using System.Collections.Generic; using System.Linq; using System.Threading; using System.Threading.Tasks; using Microsoft.Extensions.Diagnostics.HealthChecks; using Microsoft.Extensions.Options;  namespace FeedbackService.Api.HealthCheck {  public class MemoryHealthCheck : IHealthCheck  {  private readonly IOptionsMonitor<MemoryCheckOptions> \_options;   public MemoryHealthCheck(IOptionsMonitor<MemoryCheckOptions> options)  {  \_options = options;  }   public string Name => "memory\_check";   public Task<HealthCheckResult> CheckHealthAsync(  HealthCheckContext context,  CancellationToken cancellationToken = default(CancellationToken))  {  var options = \_options.Get(context.Registration.Name);   // Include GC information in the reported diagnostics.  var allocated = GC.GetTotalMemory(forceFullCollection: false);  var data = new Dictionary<string, object>()  {  { "AllocatedBytes", allocated },  { "Gen0Collections", GC.CollectionCount(0) },  { "Gen1Collections", GC.CollectionCount(1) },  { "Gen2Collections", GC.CollectionCount(2) },  };  var status = (allocated < options.Threshold) ? HealthStatus.Healthy : HealthStatus.Unhealthy;   return Task.FromResult(new HealthCheckResult(  status,  description: "Reports degraded status if allocated bytes " +  $">= {options.Threshold} bytes.",  exception: null,  data: data));  }  }  public class MemoryCheckOptions  {  public string Memorystatus { get; set; }  //public int Threshold { get; set; }  // Failure threshold (in bytes)  public long Threshold { get; set; } = 1024L \* 1024L \* 1024L;  } } |

This health check evaluates the memory status of the Feedback Service based on the allocated bytes.

Now let’s configure **RemoteHealthCheck.cs** and **MemoryHealthCheck.cs inside the HealthCheck.cs**

**HealthCheck.cs**

|  |
| --- |
| public static void ConfigureHealthChecks(this IServiceCollection services,IConfiguration configuration)  {  services.AddHealthChecks()  .AddSqlServer(configuration["ConnectionStrings:Feedback"], healthQuery: "select 1", name: "SQL servere", failureStatus: HealthStatus.Unhealthy, tags: new[] { "Feedback", "Database" })  .AddCheck<RemoteHealthCheck>("Remote endpoints Health Check", failureStatus: HealthStatus.Unhealthy)  .AddCheck<MemoryHealthCheck>($"Feedback Service Memory Check", failureStatus: HealthStatus.Unhealthy, tags: new[] { "Feedback Service" })  .AddUrlGroup(new Uri("https://localhost:44333/api/v1/heartbeats/ping"), name: "base URL", failureStatus: HealthStatus.Unhealthy);   //services.AddHealthChecksUI();  services.AddHealthChecksUI(opt =>  {  opt.SetEvaluationTimeInSeconds(10); //time in seconds between check   opt.MaximumHistoryEntriesPerEndpoint(60); //maximum history of checks   opt.SetApiMaxActiveRequests(1); //api requests concurrency   opt.AddHealthCheckEndpoint("feedback api", "/api/health"); //map health check api    })  .AddInMemoryStorage();  } |

**outputs:**Endpoint: */api/health*

A screenshot of a computer program

Description automatically generated

Endpoint: */healthcheck-ui*

A screenshot of a computer

Description automatically generated

So, there you have it! We’ve successfully implemented a few health checks inside the web API. With these checks in place, your application is now equipped to monitor and ensure its well-being. Keep building robust and resilient applications with the power of health checks in .NET 8!