**Мониторинг работоспособности приложения**

**Health Check Middleware**

Любое приложение не защищено от ошибок и снижения работоспособности в силу ряда причин, некоторые из которых трудно спрогнозировать на этапе разработки. И соответственно возникает вопрос мониторинга рабоспособности приложения. Фреймворк ASP.NET Core предоставляет специальный компонент middleware для отслеживания работоспособности приложения. С помощью данного middleware мы можем настроить проверку различных метрик и показателей, которые нам важны в рамках конкретного приложения. Например, можно проверять доступность какого-то сетевого сервиса, базы данных, использование физических ресурсов сервера (памяти, диска и т.д.)

### Добавление Health Checks Middleware

Для добавления функционала проверки работоспособности прежде всего необходимо добавить в коллекцию сервисов приложения сервис HealthCheckService с помощью метода **AddHealthChecks()**

|  |  |
| --- | --- |
| 1 | builder.Services.AddHealthChecks(); |

Проверка работоспособности доступна через специальные конечные точки. Для задания конечной точки применяются два метода:

|  |  |
| --- | --- |
| 1 | app.UseHealthChecks("/health"); |

и

|  |  |
| --- | --- |
| 1 | app.MapHealthChecks("/health"); |

В реальности разница между этими не большая. Первый метод принимает, как минимум, строку пути, запрос по которому будет обрабатываться. Второй метод принимает шаблон пути. В случаях выше оба метода позволяют обрабатывать запросы по пути "/health"

### Пример применения Health Check Middleware

Рассмотрим небольшой пример, где имитириуется применение этой функционость. Допустим, наше приложение разделено на два слоя. Первый слой (первое приложение) генерирует некоторые данные, а второй слой (второе приложение) получает данные и непосредственно взаимодействует с клиентом.

В первом проекте, который будет отвечать за генерацию данных (допустим, он будет называться **DataApp**) определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | var builder = WebApplication.CreateBuilder(args);  builder.WebHost.UseUrls("https://[::]:33333");    var app = builder.Build();    app.MapGet("/reset", () =>  {      Latency.ResetLatency();      return "Application reset";  });  app.MapGet("/data", async () =>  {      int latency = Latency.GetLatency();      await Task.Delay(latency);      return $"Application latency: {latency}";  });    app.Run();    static class Latency  {      static int counter = 1;      // увеличиваем счетчик      public static int GetLatency() => counter++ \* 500;      // сбрасываем счетчик      public static void ResetLatency() => counter = 1;  } |

Это приложение будет имитировать латентность или задержки при получении и обработке запросов. Для этого определяем вспомогательный статический класс Latency. Его метод GetLatency() увеличивает счетчик и возвращает значение counter++ \* 500. Метод ResetLatency() сбрасывает значение счетчика к начальному.

Для простоты данное приложение будет запускаться по адресу https://localhost:33333.

Приложение определяет две конечные точки. Конечная точка app.MapGet("/reset"... обрабатывает запросы по пути "reset" и условно осуществляет восстановление сервера (по сути сбрасывает значение счетчика к начальному).

Вторая конечная точка - app.MapGet("/data"... собственно посылает данные. Но для имитации все повыщаеющеся латентности приложения ее обработчик получает новое значение из метода Latency.GetLatency(), осуществляет задержку и отправляет ответ.

|  |  |
| --- | --- |
| 1  2  3 | int latency = Latency.GetLatency();  await Task.Delay(latency);  return $"Application latency: {latency}"; |

То есть таким образом, мы имитируем повышение латентности с каждым новым запросом. Соответственно с каждым новым запросом при вызове метода Latency.GetLatency() будет все больше увеличиваться значение счетчика и будет возвращаться все большее значение. И сервер будет все медленнее и медленнее обрабатывать запросы.

Теперь определим второй проект ASP.NET Core, который будет обращаться к предыдущему приложению DataApp и проверять его работоспособность:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | using Microsoft.Extensions.Diagnostics.HealthChecks;  using System.Diagnostics;    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddHealthChecks()      .AddCheck<RequestTimeHealthCheck>("RequestTimeCheck");  // проверяем работоспособность с RequestTimeCheck    builder.Services.AddHttpClient();   // подключаем HttpClient  builder.WebHost.UseUrls("https://[::]:44444");  // обрабатываем запросы по адресу [https://localhost:44444](https://localhost:44444/)    var app = builder.Build();  app.MapHealthChecks("/health");    app.MapGet("/", async (HttpClient httpClient) =>  {      // отправляем запрос к другому сервису и возвращаем его ответ      var response = await httpClient.GetAsync("<https://localhost:33333/data>");      return await response.Content.ReadAsStringAsync();  });    app.Run();    public class RequestTimeHealthCheck : IHealthCheck  {      int degraded\_level = 2000;  // уровень плохой работы      int unhealthy\_level = 5000; // нерабочий уровень      HttpClient httpClient;      public RequestTimeHealthCheck(HttpClient client) => httpClient = client;      public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,          CancellationToken cancellationToken = default)      {          // получаем время запроса          Stopwatch sw = Stopwatch.StartNew();          await httpClient.GetAsync("<https://localhost:33333/data>");          sw.Stop();          var responseTime = sw.ElapsedMilliseconds;          // в зависимости от времени запроса возвращаем определенный результат          if (responseTime < degraded\_level)          {              return HealthCheckResult.Healthy("Система функционирует хорошо");          }          else if (responseTime < unhealthy\_level)          {              return HealthCheckResult.Degraded("Снижение качества работы системы");          }          else          {              return HealthCheckResult.Unhealthy("Система в нерабочем состоянии. Необходим ее перезапуск.");          }      }  } |

Здесь прежде всего подключаем сервис проверки работоспособности:

|  |  |
| --- | --- |
| 1 | builder.Services.AddHealthChecks() |

Этот метод возвращает объект IHealthCheckBuilder, который применяется для создания и настройки сервиса HealthCheckService. Но сама проверка работоспособоности применяется объект IHealthCheck. И для добавления такого объекта применяется метод **AddCheck()**:

|  |  |
| --- | --- |
| 1 | .AddCheck<RequestTimeHealthCheck>("RequestTimeCheck"); |

Этот метод типизируется типом, который реализует интерфейс IHealthCheck, а в качества параметра принимает строку - имя для сервиса проверки.

В нашем случае в качестве реализации IHealthCheck применяется класс RequestTimeCheck. Класс должен реализовать метод интерфейса **CheckHealthAsync**. В примере выше в этом методе отправляем запрос к первому приложению к его конечной точке "/data" и проверяем время запроса:

|  |  |
| --- | --- |
| 1  2  3  4 | Stopwatch sw = Stopwatch.StartNew();  await httpClient.GetAsync("<https://localhost:33333/data>");  sw.Stop();  var responseTime = sw.ElapsedMilliseconds; |

Если время запроса превосходит определенные предустоновленые пределы, то возвращает соответствующее сообщение о работоспособности приложения:

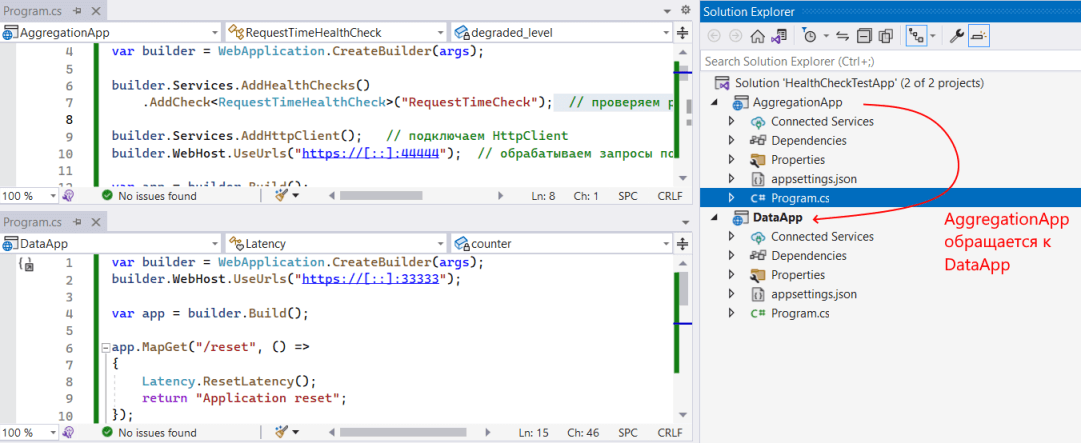
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if (responseTime < degraded\_level)  {      return HealthCheckResult.Healthy("Система функционирует хорошо");  }  else if (responseTime < unhealthy\_level)  {      return HealthCheckResult.Degraded("Снижение качества работы системы");  }  else  {      return HealthCheckResult.Unhealthy("Система в нерабочем состоянии. Необходим ее перезапуск.");  } |

Метод возвращает результат проверки работоспособности - структура HealthCheckResult. Конкретный результат устанавливается с помощью одного из методов структуры: Healthy() (приложение работает нормально), Degraded() (работоспособность снижается) и Unhealthy() (приложение неработоспособно). Каждый метод возвращает соответствующий экземпляр структуры, который сигнализирует о состоянии приложения.

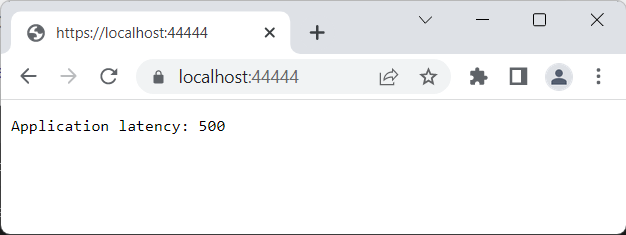
Когда, при каких условиях считать приложение неработоспособным - это зависит от нашей задачи, логики нашего приложения, метрик, которые мы применяем для оценки и конкретной ситуации. В данном случае все зависит от времени запроса.

Для получения информации о работоспособности приложении определена конечная точка "/health" посредством метода

|  |  |
| --- | --- |
| 1 | app.MapHealthChecks("/health"); |

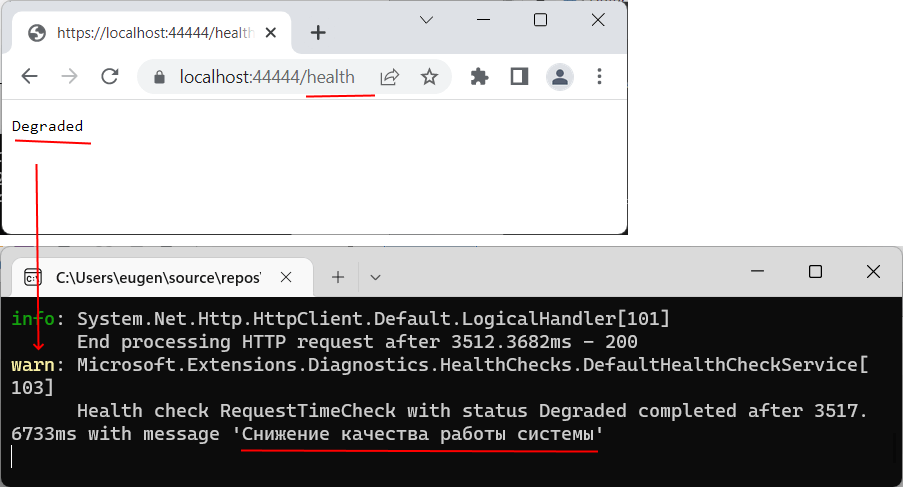


Запустим сначала приложение DataApp, а затем AggregationApp. Обратимся в браузере по адресу "https://localhost:44444/" (то есть к AggregationApp):

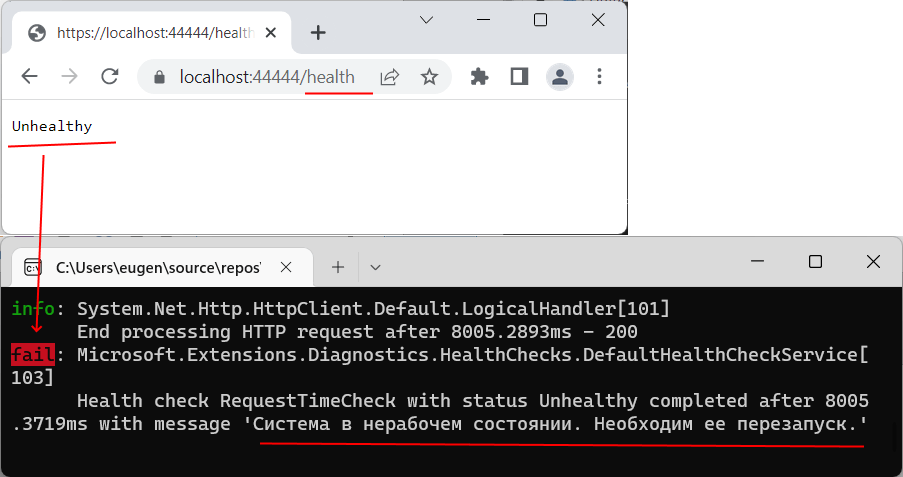


В данном случае AggregationApp будет обращаться по адресу "https://localhost:33333/data" к DataApp и получает данные. Но внутри DataApp это приведет к увеличению задержки при обработки запроса. И каждый последующий запрос будет обрабатываться все медленнее и медленнее.

Если после 5 запросов мы обратимся в браузере по адресу "https://localhost:44444/health", то консоль браузера выведет предупреждение о снижении работоспособности



Еще после нескольких запросов приложение сигнализирует об условно нерабочем состоянии, что свидетельствует, что латентность в приложении DataApp превысила сколь-нибудь допустимые пределы.



Таким образом, на основании некоторых метрик мы можем определить мехнизм уведомления о состоянии системы.

### Сервис мониторинга

Следует отметить, что, как правило, за мониторинг приложения отвечает какое-то внешнее приложение. Такие приложения мониторинга еще называют "watchdog" (дословно "сторожевой пёс", но в русскоязычной литературе для этого обычно используется понятие "Сторожевой таймер"). Так, в примере выше и DataApp и AggregationApp можно рассматривать как слои/уровни одного общего приложения. Для мониторинга определим третий проект. Если речь идет о C#, то нередко для этой цели определяется фоновый сервис. Но для простоты и текста мы определим простое консольное приложение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | HttpClient client = new HttpClient();    while (true)  {      using var response = await client.GetAsync("<https://localhost:44444/health>");      var status = await response.Content.ReadAsStringAsync();      if (status == "Unhealthy")      {          Console.WriteLine($"{DateTime.Now.ToLongTimeString()} : сервер в нерабочем состоянии, осуществляется перезапуск.");          await client.GetAsync("<https://localhost:33333/reset>");      }      else      {          Console.WriteLine($"{DateTime.Now.ToLongTimeString()} : все норм");      }      await Task.Delay(10000);   // задержка на 10 секунд  } |

В данном случае сначала осуществляем запрос по адресу "https://localhost:44444/health" и получаем статус. Если приложение в нерабочем состоянии, то обращаемся по адресу "https://localhost:33333/reset" и условно перезапускаем приложение DataApp (фактически сбрасываем счетчик, что увеличивает скорость обработки запросов).

