УДК 004

Автоматизация регулярных запросов в веб-сервисах на ASP.NET при помощи Hangfire

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассматриваются библиотеки, которые позволяют автоматизировать процесс вызова определённой функции через определённые промежутки времени. В статье также сравниваются 2 библиотеки путём решения одной простой задачи и приводится пример использования библиотеки Hangfire в приложении, работающем на ASP.NET.
Ключевые слова: C#, Hangfire, Quartz.NET, ASP.NET, автоматизация процессов, паттерны проектирования

Automating regular queries in ASP.NET web services with Hangfire

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article discusses libraries that allow you to automate the process of calling a certain function at certain intervals. The article also compares 2 libraries by solving one simple task and gives an example of using Hangfire library in an application running on ASP.NET.
Keywords: C#, Hangfire, Quartz.NET, ASP.NET, process automation, design patterns

1. Введение

1.1 Актуальность

Описанная в данной статье тема является в настоящее время очень актуальной, так как оптимизированных методов создания регулярных вызовов частей кода до сих пор остаётся важной частью создания приложений и веб-сервисов. Библиотек, созданных для выполнения такой задачи, в настоящее время существует много, но не все из них предоставляют хорошую оптимизацию и простоту применения.

Суть проблемы состоит в следующем: во многих приложениях и веб-решениях есть необходимость автоматизировать показ определённых элементов и уведомлений (например, вывод сообщений в трею уведомлений телефона) в определённое время. От контекста задачи время может варьироваться от минуты до месяцев.

Можно использовать простые методы реализации подобных возможностей через встроенные функции ожидания в приложении, но это не является оптимальным способом и может сильно нагружать приложение, если регулярных функций будет слишком много. Поэтому были созданы библиотеки, которые на английском называются Scheduler (Планировщик), например, Hangfire, Quartz.NET и другие.

1.2 Обзор исследований

Исследований в этой области очень мало. Получилось выделить только одну работу по представленной теме.

V.A. Kozhevnikov и E.S Pankratova в своей работе описали процесс разработки чат-бота в Telegram [1]. В качестве планировщика был выбран Hangfire. Бот обладает широким функционалом: работа с тикетами, файлами и т.д.

1.3 Цель исследования

Цель – рассмотреть библиотеки автоматизации вызова функций в приложениях, созданных на C#.

2. Материалы и методы

Для реализации поставленной цели используется среда разработки Visual Studio 2022, C# и Hangfire.

3. Результаты и обсуждения

В программировании часто приходится сталкиваться с задачами, которые должны выполняться регулярно (не путать с регулярными выражениями). Такие процессы зачастую автоматизированы на уровне поочерёдного выполнения задач в определённый промежуток времени.

В пример можно привести генерацию отчётов. Отчёты могут содержать информацию различного характера, например, данные о работе пользователя в системе, о работе сервера, загруженности БД. Аналогом таким процессам в SQL являются триггеры и процедуры. Такие объекты полноценной экосистемы предназначены для автоматизации процессов изменения данных в БД после выполнения определённых действий над таблицами.

В настоящее время любой ЯП поддерживает регулярный вызов функций и процедур в конкретный промежуток времени. Например, в мире Python есть большое количество модулей по работе с фоновыми задачами:

1. Celery: Celery является одним из наиболее популярных фреймворков для распределенного выполнения задач в фоновом режиме. Он поддерживает асинхронную обработку задач, планирование периодических задач, масштабирование и многое другое. Celery предоставляет гибкую конфигурацию и интеграцию с различными брокерами сообщений, такими как RabbitMQ или Redis.
2. Huey: Huey - легкий и простой в использовании фреймворк для планирования и выполнения фоновых задач. Он предоставляет простой API

для определения и запуска задач, поддерживает планирование периодических задач и имеет интеграцию с Redis в качестве брокера сообщений. Huey обеспечивает надежное выполнение задач и масштабируется с помощью запуска нескольких рабочих процессов.

3. APScheduler: APScheduler предоставляет возможность планирования и выполнения периодических задач в фоновом режиме. Он позволяет определить задачи, которые будут запускаться по расписанию, используя различные стратегии планирования (например, по времени, по календарю или с использованием крон-выражений). APScheduler может работать с различными бэкэндами, такими как базы данных или сообщения MQTT.

4. Dramatiq: Dramatiq - простой и надежный фреймворк для обработки фоновых задач. Он предоставляет удобный API для создания и запуска задач, поддерживает асинхронное выполнение, периодические задачи и масштабирование с помощью распределенной архитектуры. Dramatiq также интегрируется с RabbitMQ, Redis и другими брокерами сообщений.

Не является исключением и C#. В данном языке программирования есть библиотека Hangfire.

Hangfire - библиотека для платформы ASP.NET, предназначенная для управления фоновыми задачами и планирования их выполнения. Она позволяет легко создавать и запускать задачи, которые выполняются в фоновом режиме, без необходимости держать открытыми сеансы браузера или ожидать завершения длительных операций.

Главной целью Hangfire является автоматизация выполнения фоновых задач в веб-приложениях ASP.NET. Он позволяет отделить процессы, которые требуют длительного времени выполнения, от основного запроса пользователя, что улучшает отзывчивость приложения и позволяет обрабатывать большое количество задач параллельно. Hangfire также предоставляет надежный механизм для выполнения периодических задач, например, отправки писем, обновления кэша или синхронизации данных.

Основные процессы, которые автоматизирует Hangfire:

1. Фоновые задачи: Hangfire позволяет определить и запустить задачи, которые будут выполняться асинхронно в фоновом режиме. Это может быть любая операция, которая требует времени, например, обработка длительных вычислений, генерация отчетов или обновление базы данных.

2. Периодические задачи: Hangfire предоставляет возможность запуска задач по расписанию. Вы можете определить задачу, которая будет выполняться через определенные промежутки времени, например, каждый день, каждую неделю или каждый месяц. Это полезно для автоматизации регулярных операций, таких как рассылка уведомлений или очистка старых данных.

Можно выделить следующие преимущества Hangfire перед аналогами (Quartz.NET, FluentScheduler, NServiceBus и т. д.):

1. Простота использования: Hangfire обладает простым и интуитивно понятным API, что упрощает его интеграцию в существующие проекты

ASP.NET. Он предлагает набор готовых инструментов для создания, планирования и отслеживания фоновых задач.

2. Масштабируемость: Hangfire разработан с учетом масштабируемости и позволяет выполнять множество задач параллельно, распределяя нагрузку между несколькими процессами или серверами. Это особенно полезно при обработке больших объемов данных или при необходимости обработки задач в режиме реального времени.

3. Надежность: Hangfire обеспечивает надежное выполнение задач. Он использует надежные хранилища данных (например, SQL Server, Redis), чтобы гарантировать сохранность задач даже при сбоях системы. Если приложение перезагружается или происходит сбой, Hangfire сохраняет состояние задач и возобновляет их выполнение после восстановления работы.

4. Расширяемость: Hangfire предоставляет возможность создания собственных расширений и добавления дополнительной функциональности в систему фоновых задач. Вы можете создавать собственные обработчики задач, реализовывать собственные фильтры и использовать различные методы сериализации данных.

Установка Hangfire не является сложным процессом. Есть 2 варианта, позволяющие установить библиотеку в проект:

1. Использовать NuGet Packet Manager (см. рис. 1).
2. Использовать консоль диспетчера пакетов (см. рис. 2).

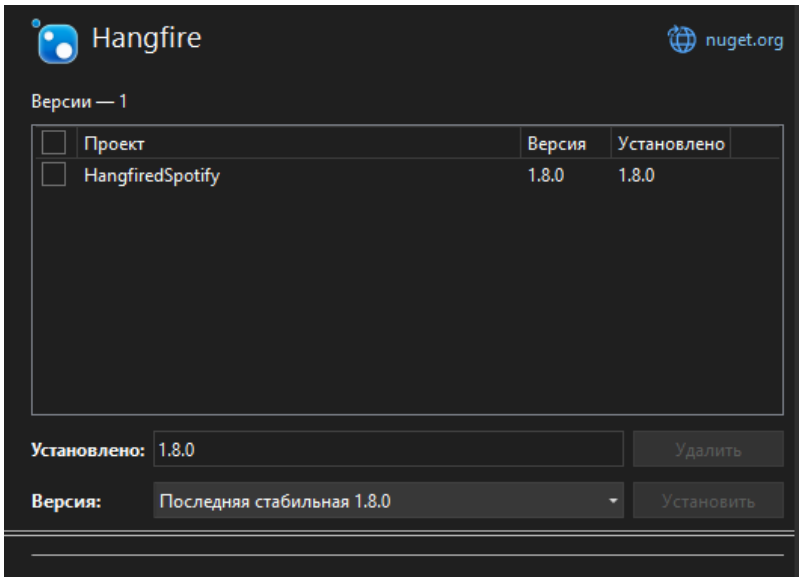

Рисунок 1. Hangfire в NuGet Packer Manager

В случае с консолью всё также легко: достаточно ввести одну команду, и библиотека будет установлена:

```
Install-Package Hangfire
```

Рисунок 2. Процесс установки Hangfire через консоль

В данной статье будут рассматриваться преимущества Hangfire на примере решения простой задачи и сравнения полученного кода с решениями на Quartz.NET. Также будет решена одна задача с использованием API сервиса стриминга музыкального контента Spotify.

Первая задача, которую необходимо решить: автоматизировать отправку уведомлений в консоль пользователя (легко адаптируется и под ASP.NET).

Первым делом необходимо создать класс, в котором будут храниться функции для работы с уведомлениями. В контексте задачи достаточно одной функции – отправки сообщений пользователю:

```
public class Notifications
{
public void SendNotification(string message)
     {
          Console.WriteLine("Отправка уведомления: " + message);
     }
}
```

Этот класс будет актуален для обеих версий решения поставленной задачи, поэтому менять под Quartz.NET его нет необходимости. В случае с Hangfire, следующим этапом будет создание функций настройки Hangfire в скрипте запуска программы. Важным шагом в написании процессов на Hangfire является то, что библиотеке необходимо подключение к БД. Для этого нужно получить строку подключения и передать её в настройки Hangfire. Чтобы получить строку подключения, необходимо установить на ПК SQL Server Management и перейти к БД в Visual Studio. В свойствах БД можно будет найти строку подключения.

Код подключения к Hangfire будет выглядеть следующим образом:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHangfire(config =>
config.UseSqlServerStorage(_configuration.GetConnectionString("D
efaultConnection")));
}
public void Configure(IApplicationBuilder app, IServiceProvider
serviceProvider)
{
    app.UseHangfireDashboard();
    app.UseHangfireServer();
```

```
    var notificationService =
serviceProvider.GetService<Notifications>();
    BackgroundJob.Enqueue(() =>
notificationService.SendNotification("Уведомление"));
}
```

Методы, необходимые для подключения к Hangfire, определены в функции Configure: UseHangfireDashboard и UseHangfireServer. Перед этим в Hangfire передаётся строка подключения через метод AddHangfire. В этом методе указывается хранилище SQL через функцию UseSqlServerStorage и метод класса Configuration, а именно GetConnectionString.

Также в Configure определяются и другие параметры приложения, если в этом есть необходимость.

Функция Enqueue класса BackgroundJob как раз создаёт регулярный вызов процедуры. Есть также другой способ определения регулярных запросов в приложении, который больше подходит для ASP.NET. Таким образом была создан регулярный вызов функции с помощью Hangfire.

Теперь стоит рассмотреть реализацию на Quartz.NET. Для начала его нужно установить через команду в консоли:

```
Install-Package Quartz
```

После этого также создаётся класс уведомлений, который был описан выше. Следующим этапом можно выделить создание класса и метода, где будет определена периодическая задача Quartz.NET:

```
public class NotificationJob : IJob
{
    public void Execute(IJobExecutionContext context)
    {
        var notificationService = new Notifications();
        notificationService.SendNotification("Пример
уведомления");
    }
}
```

После этого также нужно изменить стартовый скрипт приложения и добавить настройки в приложение:

```
public void ConfigureServices(IServiceCollection services)
{
services.AddSingleton<IJobFactory, SingletonJobFactory>();
services.AddSingleton<ISchedulerFactory, StdSchedulerFactory>();
services.AddSingleton<NotificationJob>();

services.AddHostedService<QuartzHostedService>();
}
```

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
     app.ApplicationServices.GetService<QuartzHostedService>().St
artAsync(CancellationToken.None).Wait();
}
```

Тут необходимо понимать принцип работы синглтонов, что уже является недостатком, если с приложением работает начинающий разработчик. В коде выше объявлены классы SingletonJobFactory (является шаблоном в C#) и QuartzHostedService, но при этом они нигде не описаны. Необходимо создать эти классы:

```
public class SingletonJobFactory : IJobFactory
{
    private readonly IServiceProvider _serviceProvider;

    public SingletonJobFactory(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public IJob NewJob(TriggerFiredBundle bundle, IScheduler
scheduler)
    {
        return
_serviceProvider.GetRequiredService(bundle.JobDetail.JobType) as
IJob;
    }

    public void ReturnJob(IJob job)
    {
    }
}
```

Этот класс является фабрикой для создания экземпляров IJob, используемых Quartz.NET для выполнения задач. В стандартной конфигурации Quartz.NET создает новый экземпляр IJob для каждого выполнения задачи. Однако в ASP.NET приложении желательно использовать механизм внедрения зависимостей (Dependency Injection), чтобы иметь возможность использовать зависимости и сервисы, зарегистрированные в контейнере внедрения зависимостей. SingletonJobFactory позволяет внедрять зависимости в IJob путем получения экземпляра IJob из контейнера сервисов ASP.NET. Таким образом, зависимости могут быть инъецированы в IJob, что позволяет использовать другие сервисы и компоненты вашего приложения внутри задачи Quartz.NET.

Также необходимо создать QuartzHostedService:

```
public class QuartzHostedService : IHostedService
{
    private readonly IScheduler _scheduler;

    public QuartzHostedService(ISchedulerFactory
schedulerFactory, IEnumerable<NotificationJob> jobs)
    {
        _scheduler = schedulerFactory.GetScheduler().Result;

        foreach (var job in jobs)
        {
            var jobDetail = JobBuilder.Create(job.GetType())
                .WithIdentity(job.GetType().FullName)
                .Build();

            var trigger = TriggerBuilder.Create()

.WithIdentity($"{job.GetType().FullName}.trigger")
                .StartNow()
                .WithSimpleSchedule(x =>
x.WithInterval(TimeSpan.FromSeconds(10)).RepeatForever())
                .Build();

            _scheduler.ScheduleJob(jobDetail, trigger).Wait();
        }
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        return _scheduler.Start();
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return _scheduler.Shutdown();
    }
}
```

Этот класс реализует интерфейс IHostedService и используется для интеграции Quartz.NET в жизненный цикл вашего ASP.NET приложения. QuartzHostedService отвечает за запуск и остановку планировщика Quartz.NET. При запуске приложения QuartzHostedService создает экземпляр планировщика Quartz.NET, регистрирует все необходимые задачи (jobs) и запускает планировщик. При остановке приложения QuartzHostedService останавливает планировщик Quartz.NET. Таким образом, QuartzHostedService гарантирует, что задачи Quartz.NET будут выполняться внутри вашего ASP.NET приложения и будут управляться его жизненным циклом.

Как можно заметить, код на Hangfire получился компактнее. Его легче читать, отлаживать и он легко масштабируется. В случае с Quartz.NET есть

проблемы: нужно понимать шаблоны проектирования, знать про хост-сервисы и принцип построения сложных ИС.

В ряде характеристик Hangfire уступает Quartz.NET, но при этом остаётся простым в использовании и настройке.

Таблица 1. Сравнение характеристик библиотек Hangfire и Quartz.NET

Характеристика	**Hangfire**	**Quartz.NET**
Управление фоновыми задачами	Да	Да
Типы задач	Поддержка различных типов задач (методы, классы, лямбда-выражения и т.д.)	Поддержка различных типов задач (классы, методы)
Мониторинг и управление	Панель управления и мониторинг задач	Отсутствует встроенная панель управления, доступ к API для мониторинга и управления
Внедрение зависимостей	Поддержка внедрения зависимостей через DI-контейнеры	Отсутствует поддержка внедрения зависимостей
Поддержка персистентности	Различные провайдеры хранения данных (SQL Server, Redis и др.)	Поддержка различных провайдеров хранения данных (SQL Server, Oracle, MySQL и др.)
Хранилище задач	SQL Server (по-умолчанию), ряд СУБД через расширения, Redis (в платной версии)	In-memory, ряд БД (SQL Server, MySQL, Oracle...)
Web-интерфейс	Да	Нет
Реализация многопоточности	TSL	Thread, Monitor

Также Hangfire может работать и с ASP.NET проектами. В качестве примера можно взять приложения, которое каждый день будет выводить список композиций из плейлиста пользователя Spotify.

Для выполнения данной задачи также потребуется библиотека RestSharp:

```
Install-Package RestSharp
```

После необходимо создать класс, который будет получить информацию о плейлистах:

```
public void UpdatePlaylistData()
{
    var playlistId = "playlistid ";
    var accessToken = "accesstoken ";

    var client = new
RestClient("https://api.spotify.com/v1/playlists/" +
playlistId);
    var request = new RestRequest(Method.GET);
    request.AddHeader("Authorization", "Bearer " + accessToken);
    IRestResponse response = client.Execute(request);
}
```

Остаётся добавить одну функцию в основную часть программы:

```
RecurringJob.AddOrUpdate("update-playlist-data", () =>
UpdatePlaylistData(), Cron.Daily);
```

Настройки подключения задаются также, как и было описано ранее, через код основной части программы.

Таким образом, в данной работе были рассмотрены примеры использования библиотеки Hangfire. Также были рассмотрены преимущества перед конкурентами и было проведено сравнение с существующим аналогом. В конце работы рассмотрен пример использования библиотеки в приложении, работающем на фреймворке ASP.NET.

Библиографический список

1. Kozhevnikov V.A., Pankratova E.S. The customer support service development for user applications // Theoretical & Applied Science. 2019. № 4 (72). С. 352-363.
2. Hangfire – Background jobs and workers for .NET URL: https://www.hangfire.io (дата обращения 18.05.2023).
3. Quartz.NET: Home URL: https://www.quartz-scheduler.net (дата обращения 18.05.2023).