Documento Guia de Automatizacion de Tareas

Materia

ISW-711

Estudiante

Rafael Eduardo Alvarado Zuñiga

Conceptos Basicos

Definicion de Tareas Programadas

Una Tarea Programada es una accion automatizada que se ejecuta en un momento especifico con un intervalo especifico. Esto ayuda a automatizar procesos repetitivos, o ejecutar una tarea repetitiva sin necesidad de intervencion manual.

Proposito de las Tareas Programadas

- Automatizar procesos: Algunos ejemplos serian copias de seguridad, actualizaciones, o en este caso, envio de correos tipo recordatorio.
- Reduccion de la intervencion manual, el cual ahorra tiempo y esfuerzo.

Frecuencia de Ejecucion

Las tareas se puede colocar cualquier tipo de frecuencia, tipo diario, semanal, mensual, anual. En este caso se hara de manera diaria.

Recursos por Configurar y/o Instalar

En este caso como la automatizacion se va a realizar en una aplicacion **MVC ASP.NET Core 8**, no es necesario instalar nada extra, ya que el framework ya tiene todo lo que se necesita, nada mas se necesita realizar el codigo.

Pasos para su configuracion

Estos pasos de instalacion asumen que ya se tenga una aplicacion **ASP.NET Core 8** la misma tenga su separacion de proyectos Web, Application e Interface, cada una con el mmodelo de base de datos correctamente configurado, y los servicios necesarios ya montados, una vez teniendo esto completado se puede continuar con los pasos:

Paso 1: Creacion de dependencias necesarias para la tarea

En este ejemplo, como se va a realizar un envio de correos tipo recordatorio a los clientes, se necesita un servicio que nos ayude con el envio de correos, y se va a crear otro servicio que nos devuelva plantillas de correos para poder facilitar el proceso de llenado del body del correo.

Dependencia de Envio Correos

La dependencia de envio de correos, ya que su logica se conecta a algo externo de la aplicacion (en este caso un servidor SMTP), lo mejor es colocar la logica en el proyecto de **Infraestructure**, esto va a requerir que creemos un archivo interfaz y otro archivo que implemente dicha interfaz, ambos archivos quedaran de la siguiente forma

IRepositoryMailer.cs

```
namespace MecaAgenda.Infraestructure.Repository.Interfaces
{
    public interface IRepositoryMailer
    {
        void SendEmail(string subject, string body, List<string> to);
    }
}
```

RepositoryMailer.cs

```
using MecaAgenda.Infraestructure.Repository.Interfaces;
using System.Net.Mail;
using System.Net;

namespace MecaAgenda.Infraestructure.Repository.Implementations
{
    public class RepositoryMailer : IRepositoryMailer
    {
        private readonly string fromEmail = "";
        private readonly string emailPass = "";

        public void SendEmail(string subject, string body, List<string> to)
        {
            if (fromEmail == null || emailPass == null)
            {
                throw new Exception("Sender Credentials not found");
            }

            if (string.IsNullOrWhiteSpace(subject) ||
string.IsNullOrWhiteSpace(subject) || to.Count <= 0)
            {
                throw new Exception("Email information missing");
            }

            MailMessage message = new MailMessage();

            message.IsBodyHtml = true;
            message.Subject = subject;
            message.Body = body;
            message.From = new MailAddress(fromEmail);
            to.ForEach(x => message.To.Add(x));
```

```
            SmtpClient smtp = new SmtpClient("smtp.gmail.com");
            smtp.Port = 587;
            smtp.Credentials = new NetworkCredential(fromEmail, emailPass);
            smtp.EnableSsl = true;

            smtp.Send(message);
        }
    }
}
```

El detalle mas importante a detallar de esta implementacion es que falta la colocacion del correo y de la contraseña que se van a usar. Esta funcion de enviar el correo es sincronica, por lo que no sera necesario colocarle un **await** para esperar que esta sea completada.

Dependencia de obtencion de plantillas

Por facilidad y limpieza en el codigo, es mejor crear un servicio que extraiga el contenido de archivos **.html** para que la plantilla sea manejada en un archivo externo. Este serivicio ya que va a funcionar dentro de la aplicacion y no requiere nada externo, se va a colocar en el proyecto **Application**, y va a requerir los siguientes archivos:

IServiceMailTemplates.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MecaAgenda.Application.Services.Interfaces
{
    public interface IServiceMailTemplates
    {
        Task<string> GetEmailTemplate(string templateName);
    }
}
```

ServiceMailTemplates.cs

```
using MecaAgenda.Application.Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MecaAgenda.Application.Services.Implementations
```

```
{
    public class ServiceMailTemplates : IServiceMailTemplates
    {
        public async Task<string> GetEmailTemplate(string templateName)
        {
            var path = Path.Combine(AppContext.BaseDirectory, "email-templates",
templateName);
            return await File.ReadAllTextAsync(path);
        }
    }
}
```

Dicho codigo para que funcione, se requiere de guardar los archivos **html** en un lugar especifico, en este caso en la raiz del proyecto **Application** vamos a crear una carpeta llamada **email-templates**, dentro de dicha carpeta podemos meter plantillas **.html** que seran el cuerpo del correo, como el siguiente archivo que mostrara informacion de la cita tipo recordatorio.

appointment-reminder.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Appointment Confirmation</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            color: #333;
            line-height: 1.6;
            margin: 0;
            padding: 0;
            background-color: #f4f4f4;
        }

        .container {
            width: 80%;
            margin: 20px auto;
            background: #fff;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }

        .header {
            text-align: center;
            margin-bottom: 20px;
        }

            .header img {
                max-width: 150px;
            }
```

```
        .section {
            margin-bottom: 20px;
        }

            .section h2 {
                margin-top: 0;
                color: #007BFF;
            }

            .section p {
                margin: 5px 0;
            }

        .footer {
            text-align: center;
            font-size: 0.9em;
            color: #666;
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="header">
            <img src="" alt="Company Logo">
            <h1>Revved Up Repairs</h1>
        </div>

        <div class="section">
            <h2>Branch Information</h2>
            <p><strong>Name:</strong> {{BRANCH_NAME}}</p>
            <p><strong>Email:</strong> {{BRANCH_EMAIL}}</p>
            <p><strong>Phone:</strong> {{BRANCH_PHONE}}</p>
            <p><strong>Address:</strong> {{BRACH_ADDRESS}}</p>
        </div>

        <div class="section">
            <h2>Appointment Details</h2>
            <p><strong>Date:</strong> {{APPOINTMENT_DATE}}</p>
            <p><strong>Start Time:</strong> {{APPOINTMENT_START}}</p>
            <p><strong>Finish Time:</strong> {{APPOINTMENT_END}}</p>
            <p><strong>Type of Service:</strong> {{APPOINTMENT_TYPE}}</p>
        </div>

        <div class="footer">
            <p>&copy; 2024 Revved Up Repairs. All rights reserved.</p>
        </div>
    </div>
</body>
</html>
```

Por temas de tamaño, se removio el logo, ya que el mismo esta guardado en base64 en el archivo, pero, con este archivo se puede ver la estructura del **.html** y que tenemos campos encerrados en dos llaves (**{{ }}**), esto es para cuando se traigan los datos del **.html** sea mas facil remplazar el texto que es de la cita.

Dichos servicios deben de ser agregados a **Program.cs** para poder ser utilizados.

Paso 2: Creacion del Servicio de Segundo Plano

La creacion de este servicio es unicamente un archivo, ya que realizaremos la implementacion de una interfaz ya existente en **.NET Core 8**, en la misma estableceremos un codigo que corra de manera que corra de manera perpetua en paralelo a la aplicacion web, la misma para no enviar correos a cada rato, se le colocaran **Task.Delay** para que unicamente corra una vez al dia.

Ya que esta logica unicamente ocupa datos dentro de la aplicacion, este servicio sera creado dentro del proyecto **Application**, aunque, la implementacion de esta interfaz no ira donde siempre, en este caso en la carpeta de **Services** se creara otra carpeta dentro de la misma llamada **AutoHosted**, esto para diferencias los Servicios que podamos consumir, a los que corren por si solos. El contenido del archivo sera el siguiente:

ServiceBackground

```
using MecaAgenda.Application.Services.Interfaces;
using MecaAgenda.Infraestructure.Repository.Interfaces;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MecaAgenda.Application.Services.AutoHosted
{
    public class ServiceBackground : BackgroundService
    {
        private readonly ILogger<ServiceBackground> _logger;
        private readonly IServiceScopeFactory _serviceScopeFactory;

        public ServiceBackground(ILogger<ServiceBackground> logger,
IServiceScopeFactory serviceScopeFactory)
        {
            _logger = logger;
            _serviceScopeFactory = serviceScopeFactory;
        }

        protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
        {
            // Log that Task Schedule has been started
            _logger.LogInformation("Tasks have been started at: {time}",
DateTime.Now.ToString());
```

```
            while (!stoppingToken.IsCancellationRequested)
            {
                // Calculate next 8AM
                var now = DateTime.Now;
                var nextRunTime = now.Date.AddHours(8).AddMinutes(0);

                if (now >= nextRunTime)
                    nextRunTime.AddDays(1);

                var delay = nextRunTime - now;

                // Wait until next 8AM
                await Task.Delay(delay, stoppingToken);

                // Execute customer reminder
                await remindCustomerOfServiceAsync();
            }
        }

        private async Task remindCustomerOfServiceAsync()
        {
            // Log that the Task started
            _logger.LogInformation("Reminding customers started at: {time}",
DateTime.Now.ToString());

            using (var scope = _serviceScopeFactory.CreateScope())
            {
                var _serviceAppointment =
scope.ServiceProvider.GetRequiredService<IServiceAppointment>();
                var _repositoryMailer =
scope.ServiceProvider.GetRequiredService<IRepositoryMailer>();
                var _serviceMailTemplates =
scope.ServiceProvider.GetRequiredService<IServiceMailTemplates>();

                // Obtain Todays and Tomorrow dates
                DateOnly today =
DateOnly.Parse(DateTime.Now.Date.ToString().Split(" ")[0]);
                DateOnly tomorrow =
DateOnly.Parse(DateTime.Now.Date.AddDays(1).ToString().Split(" ")[0]);

                // Obtain the appointments for today and tomorrow
                var appointments = await _serviceAppointment.ListAsync(null, null,
today, tomorrow);

                // Create an send emails for each one of the appointments

                foreach (var appointment in appointments)
                {
                    string subject = "Reminder for Appointment at " +
appointment.Date + " from " + appointment.StartTime + " to " +
appointment.EndTime;
                    List<string> to = new List<string>(new string[] {
appointment.Client.Email });
```

```
                    string template = await
_serviceMailTemplates.GetEmailTemplate("appointment-reminder.html");

                    string body = template
                        .Replace("{{BRANCH_NAME}}", appointment.Branch.Name)
                        .Replace("{{BRANCH_EMAIL}}", appointment.Branch.Email)
                        .Replace("{{BRANCH_PHONE}}", appointment.Branch.Phone)
                        .Replace("{{BRACH_ADDRESS}}", appointment.Branch.Address)
                        .Replace("{{APPOINTMENT_DATE}}",
appointment.Date.ToString())
                        .Replace("{{APPOINTMENT_START}}",
appointment.StartTime.ToString())
                        .Replace("{{APPOINTMENT_END}}",
appointment.EndTime.ToString())
                        .Replace("{{APPOINTMENT_TYPE}}",
appointment.Service.Name);

                    try
                    {
                        _repositoryMailer.SendEmail(subject, body, to);
                    }
                    catch (Exception ex)
                    {
                        _logger.LogError("Error while sending email: {error}",
ex.Message);
                    }

                }
            }

            // Log that the Task finished
            _logger.LogInformation("Reminding customers finished at: {time}",
DateTime.Now.ToString());
        }
    }
}
```

En este caso la logica del proceso automatico reside dentro de la funcion **ExecuteAsync()**, con el **while (!stoppingToken.IsCancellationRequested)**, se crea una funcion que repite continuamente, pero gracias a los **Task.Delay()** podemos hacer que el codigo unicamente corra 1 vez por dia.

La logica del envio de correo se separa en otra funcion para asi facilmente dividir la logica que envia y la otra que automatiza la corrida del proceso. Y esta funcion de **remindCustomerOfServiceAsync()**, se puede notar que hace uso a los servicios que creamos anteriormente.

Paso 3: Adjuntar Servicio a Program.cs

Ya que el servicio, es diferente al otro tipo de servicios donde se crea su interfaz y su implementacion, este no se agrega al archivo **Program.cs** de la misma manera, para poder agregarlo, despues de que se crea el builder y se realiza el **Dependency Injection** del **Repository** y los **Services**, se agrega la siguiente lineas:

```
// Automated Service
builder.Services.AddHostedService<ServiceBackground>();
```

De esta manera el servicio se estaria agregando como un **HostedService**, y cuando se corre la solucion, dicho Servicio va a correr en paralelo.

Referencias

- Li Huan, J. (2024, May 10). Background tasks with hosted services in ASP.NET Core. Microsoft Learn. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-8.0&tabs=visual-studio