Занятие 18. Объектное хранилище

Теория

Объектное хранилище (Object Storage) — это хранилище данных в виде объектов, которые содержат данные и метаданные. Также часто его называют S3 (Simple Storage Service).

Чаще всего используется для хранения файлов, изображений, видео и других типов данных.

Популярными реализациями объектного хранилища являются:

- AWS S3;
- Azure Blob Storage;
- Google Cloud Storage;
- MinIO.

Обычно объектное хранилище имеет распределенную структуру и ориентировано на:

- хранение больших объемов неструктурированных данных;
- высокую доступность и масштабируемость;
- низкую задержку при чтении и записи данных.

Практика

Для демонстрации работы с S3 мы будем использовать MinIO - популярную реализацию S3 с открытым исходным кодом.

В качестве решаемой задачи рассмотрим создание сервиса конвертации изображений из формата PNG в JPG.

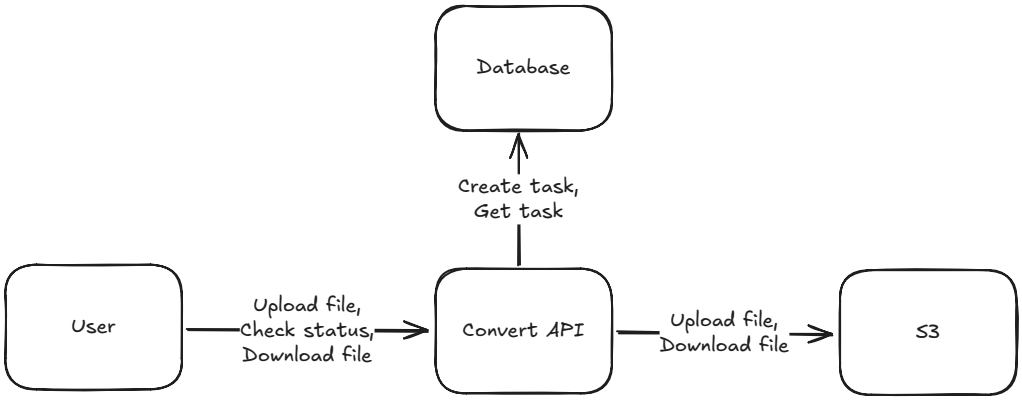
Поддерживать будем следующие сценарии:

- Пользователь загружает файл в формате PNG и получает ссылку на операцию конвертации;
- Пользователь периодически проверяет статус операции конвертации;
- Если операция завершается успешно, пользователь может скачать сконвертированное изображение;
- Если операция завершается с ошибкой, пользователь получает признак ошибки.

В данной задаче мы не будем рассматривать распределенную обработку данных, а сосредоточимся именно на работе с объектным хранилищем.

Архитектура сервиса

Архитектура нашего сервиса будет выглядеть следующим образом:



Как можно видеть, в данной архитектуре мы будем использовать следующие компоненты:

- Приложение, реализующее API;
- Базу данных, в которой будут храниться данные приложения;
- Объектное хранилище для хранения изображений.

Загрузка изображения

Сценарий загрузки изображения будет реализован следующим образом:

1. Пользователь отправляет POST-запрос на API с файлом в формате PNG.
2. Приложение сохраняет изображение в объектное хранилище.
3. Если удалось сохранить изображение, то создает задание на конвертацию изображения сохраняет его в базу данных.
4. Если удалось создать задание, то возвращает пользователю идентификатор задания.

Конвертация изображения

Сценарий конвертации изображения будет реализован следующим образом:

1. Приложение в фоне проверяет наличие заданий на конвертацию изображений.
2. Если находит, то скачивает изображение из объектного хранилища.
3. Если удалось скачать изображение, то конвертирует его в JPG.
4. Если удалось конвертировать изображение, то сохраняет его в объектное хранилище.
5. Если удалось сохранить изображение, то добавляет идентификатор сконвертированного изображения в базу данных.

Скачивание изображения

Сценарий скачивания изображения будет реализован следующим образом:

1. Пользователь отправляет GET-запрос на API с идентификатором задания.
2. Приложение проверяет наличие сконвертированного изображения в объектном хранилище.
3. Если изображение найдено, то происходит скачивание изображения.
4. Если изображение не найдено, то возвращается ошибка.

Реализация

Подготовка инфраструктуры

Начнем с того, что сразу же развернем S3 и базу данных. Для этого будем использовать Docker Compose.

Создадим каталог для нашего проекта и перейдем в него:

```
mkdir ImageConverter
cd ImageConverter
```

В каталоге создадим файл `docker-compose.yml` и добавим в него следующий код:

```
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: image-converter
      POSTGRES_USER: postgres
    ports:
      - 5432:5432
  minio:
    image: minio/minio
    expose:
      - 9000
      - 9001
    volumes:
      - minio-data:/data
    environment:
      MINIO_ROOT_USER: "minioadmin"
      MINIO_ROOT_PASSWORD: "minioadmin"
    command: server /data --console-address ":9001"
  createbuckets:
    image: minio/mc
    depends_on:
      - minio
    entrypoint: >
      /bin/sh -c "
      /usr/bin/mc alias set myminio http://minio:9000 minioadmin minioadmin;
      /usr/bin/mc rm -r --force myminio/image-converter;
      /usr/bin/mc mb myminio/image-converter;
      exit 0;
      "
volumes:
  minio-data:
```

Как можно видеть, кроме контейнера с базой данных и S3, мы также запускаем контейнер, который автоматически создает бакет `image-converter` в S3 при запуске.

Теперь запустим Docker Compose:

```
docker compose up -d
```

Создание проекта

Когда у нас есть инфраструктура, мы можем приступить к созданию приложения.

Для простоты приложение будет реализовано в виде единого проекта без разделения на слои.

Создадим решение и файлы проекта:

```
dotnet new sln -n ImageConverter
dotnet new webapi -n ImageConverter.WebApi
dotnet sln ImageConverter.sln add ImageConverter.WebApi
```

Сразу же добавим в проект необходимые зависимости. В нашем случае это:

- EF Core - для работы с базой данных;
- S3 SDK - для работы с объектным хранилищем;
- SkiaSharp - для работы с изображениями.

Для этого выполним следующие команды:

```
dotnet add ImageConverter.WebApi package Microsoft.EntityFrameworkCore.Design --
version 8
dotnet add ImageConverter.WebApi package Npgsql.EntityFrameworkCore.PostgreSQL --
version 8
dotnet add ImageConverter.WebApi package AWSSDK.S3 --version 4
dotnet add ImageConverter.WebApi package SkiaSharp
dotnet add ImageConverter.WebApi package
SkiaSharp.NativeAssets.Linux.NoDependencies
dotnet add ImageConverter.WebApi package Swashbuckle.AspNetCore --version 8
dotnet add ImageConverter.WebApi package Swashbuckle.AspNetCore.SwaggerUI --
version 8
```

Как видим, кроме объявленных ранее зависимостей, мы также добавили `Swashbuckle.AspNetCore` и `Swashbuckle.AspNetCore.SwaggerUI`. В обычной ситуации этого не нужно делать, так как они уже включены в шаблон проекта. Но версия, которая поставляется по умолчанию, содержит баг, который не позволит протестировать приложение в нашем случае - поэтому мы устанавливаем пакеты вручную.

Определение моделей

Для начала определим модели, которые будут использоваться в нашем приложении. Мы будем работать с заданиями на конвертацию изображений - поэтому такую модель и заведем.

Создадим каталог `Models` в нем типы:

- `ConversionTask` - задание на конвертацию изображения;
- `ConversionTaskStatus` - статус задания на конвертацию изображения.

Сначала определим тип `ConversionTaskStatus`:

```
namespace ImageConverter.WebApi.Models;

internal enum ConversionTaskStatus
{
    Pending,
    Converting,
    Completed,
    Failed,
}
```

Далее определим тип `ConversionTask`:

```
namespace ImageConverter.WebApi.Models;

internal sealed class ConversionTask
{
    public Guid Id { get; }
    public string SourceImageId { get; }
    public string? ConvertedImageId { get; set; }
    public ConversionTaskStatus Status { get; set; }
    public DateTimeOffset CreatedAt { get; }

    public ConversionTask(Guid id, string sourceImageId, string? convertedImageId,
ConversionTaskStatus status, DateTimeOffset createdAt)
    {
        Id = id;
        SourceImageId = sourceImageId;
        ConvertedImageId = convertedImageId;
        Status = status;
        CreatedAt = createdAt;
    }
}
```

Хранение

Мы определили модели - теперь определим, как они будут храниться в базе данных.

Создадим контекст базы данных. Для этого добавим в проект каталог `Database` и в нем файл `AppDbContext.cs`:

```
namespace ImageConverter.WebApi.Database;

internal sealed class AppDbContext : DbContext
{
    public DbSet<ConversionTask> ConversionTasks { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<ConversionTask>(entity =>
        {
            entity.ToTable("conversion_tasks");

            entity.HasKey(e => e.Id);

            entity.Property(e => e.Id)
                .HasColumnName("id")
                .IsRequired();

            entity.Property(e => e.SourceImageId)
                .HasColumnName("source_image_id")
                .IsRequired();

            entity.Property(e => e.ConvertedImageId)
                .HasColumnName("converted_image_id");

            entity.Property(e => e.Status)
                .HasColumnName("status")
                .HasConversion(
                    v => v.ToString(),
                    v => Enum.Parse<ConversionTaskStatus>(v)
                )
                .IsRequired();

            entity.Property(e => e.CreatedAt)
                .HasColumnName("created_at")
                .IsRequired();
        });
    }
}
```

Далее создадим миграции. Для этого сначала определим фабрику для контекста базы данных. Сделаем это в том же каталоге:

```
namespace ImageConverter.WebApi.Database;
```

```
internal sealed class AppDbContextFactory :
IDesignTimeDbContextFactory<AppDbContext>
{
    public AppDbContext CreateDbContext(string[] args)
    {
        var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
        optionsBuilder.UseNpgsql("Host=localhost;Database=image-
converter;Username=postgres;Password=postgres");

        return new AppDbContext(optionsBuilder.Options);
    }
}
```

Теперь создадим миграции:

```
dotnet ef migrations add Initial --project ImageConverter.WebApi
```

Далее добавим сервис для запуска миграций. Для этого добавим в каталоге `Database` файл `MigrationRunner.cs`:

```
namespace ImageConverter.WebApi.Database;

internal sealed class MigrationRunner : IHostedService
{
    private readonly IServiceScopeFactory _serviceScopeFactory;

    public MigrationRunner(IServiceScopeFactory serviceScopeFactory)
    {
        _serviceScopeFactory = serviceScopeFactory;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        using var scope = _serviceScopeFactory.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

        dbContext.Database.Migrate();

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken) =>
Task.CompletedTask;
}
```

Конфигурация

Так как нам необходимо будет получать доступ к S3, заранее позаботимся о конфигурации.

Создадим в проекте каталог `Configuration` и в нем файл `S3Config.cs`:

```
namespace ImageConverter.WebApi.Configuration;

internal sealed class S3Config
{
    [Required]
    public required string ServiceUrl { get; set; }

    [Required]
    public required string AccessKey { get; set; }

    [Required]
    public required string SecretKey { get; set; }

    [Required]
    public required string BucketName { get; set; }
}
```

В данном коде мы определяем конфигурацию для S3, а именно:

- `AccessKey` - ключ доступа;
- `SecretKey` - секретный ключ;
- `ServiceUrl` - URL сервиса;
- `BucketName` - имя бакета (бакеты - это контейнеры для хранения объектов).

Загрузка изображения

Когда у нас есть контекст базы данных, мы можем приступить к реализации загрузки изображения.

Для этого в проекте создадим каталог `UseCases/UploadImage` и в нем следующие типы:

- `UploadImageRequest` - запрос на загрузку изображения;
- `UploadImageResponse` - ответ на загрузку изображения;
- `IUploadImageHandler` - интерфейс обработчика загрузки изображения;
- `UploadImageHandler` - обработчик загрузки изображения.

Сначала определим тип `UploadImageRequest`:

```
namespace ImageConverter.WebApi.UseCases.UploadImage;

internal sealed record UploadImageRequest(Stream FileContent);
```

Далее определим тип `UploadImageResponse`:

```
namespace ImageConverter.WebApi.UseCases.UploadImage;
```

```
internal sealed record UploadImageResponse(Guid TaskId);
```

Теперь определим интерфейс `IUploadImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.UploadImage;

internal interface IUploadImageHandler
{
    Task<UploadImageResponse> HandleAsync(UploadImageRequest request,
CancellationToken cancellationToken);
}
```

Далее определим класс `UploadImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.UploadImage;

internal sealed class UploadImageHandler : IUploadImageHandler
{
    private readonly IAmazonS3 _s3Client;
    private readonly S3Config _s3Config;
    private readonly AppDbContext _dbContext;

    public UploadImageHandler(IAmazonS3 s3Client, IOptions<S3Config> s3Config,
AppDbContext dbContext)
    {
        _s3Client = s3Client;
        _s3Config = s3Config.Value;
        _dbContext = dbContext;
    }

    public async Task<UploadImageResponse> HandleAsync(UploadImageRequest request,
CancellationToken cancellationToken)
    {
        var taskId = Guid.NewGuid();
        var sourceImageId = $"{taskId}-source";

        await _s3Client.PutObjectAsync(new PutObjectRequest
        {
            BucketName = _s3Config.BucketName,
            Key = sourceImageId,
            InputStream = request.FileContent,
        }, cancellationToken);

        var task = new ConversionTask(
            id: taskId,
            sourceImageId: sourceImageId,
            convertedImageId: null,
            status: ConversionTaskStatus.Pending,
            createdAt: DateTimeOffset.UtcNow
```

```
        );

        await _dbContext.ConversionTasks.AddAsync(task, cancellationToken);
        await _dbContext.SaveChangesAsync(cancellationToken);

        return new UploadImageResponse(taskId);
    }
}
```

Конвертация изображения

Мы умеем загружать файлы - теперь научимся конвертировать их. Для этого в каталоге `UseCases` создадим каталог `ConvertImage` и в нем следующие типы:

- `ConvertImageRequest` - запрос на конвертацию изображения;
- `ConvertImageResponse` - ответ на конвертацию изображения;
- `IConvertImageHandler` - интерфейс обработчика конвертации изображения;
- `ConvertImageHandler` - обработчик конвертации изображения.

Сначала определим тип `ConvertImageRequest`:

```
namespace ImageConverter.WebApi.UseCases.ConvertImage;

internal sealed record ConvertImageRequest(Guid TaskId);
```

Далее определим тип `ConvertImageResponse`:

```
namespace ImageConverter.WebApi.UseCases.ConvertImage;

internal sealed record ConvertImageResponse(bool Success);
```

Теперь определим интерфейс `IConvertImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.ConvertImage;

internal interface IConvertImageHandler
{
    Task<ConvertImageResponse> HandleAsync(ConvertImageRequest request,
CancellationToken cancellationToken);
}
```

Далее определим класс `ConvertImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.ConvertImage;

internal sealed class ConvertImageHandler : IConvertImageHandler
{
    private readonly IAmazonS3 _s3Client;
    private readonly S3Config _s3Config;
    private readonly AppDbContext _dbContext;
    private readonly ILogger<ConvertImageHandler> _logger;

    public ConvertImageHandler(IAmazonS3 s3Client, IOptions<S3Config> s3Config,
AppDbContext dbContext, ILogger<ConvertImageHandler> logger)
    {
        _s3Client = s3Client;
        _s3Config = s3Config.Value;
        _dbContext = dbContext;
        _logger = logger;
    }

    public async Task<ConvertImageResponse> HandleAsync(ConvertImageRequest
request, CancellationToken cancellationToken)
    {
        var task = await _dbContext.ConversionTasks.FindAsync([request.TaskId],
cancellationToken);
        if (task == null)
        {
            throw new InvalidOperationException($"Task {request.TaskId} not
found");
        }

        try
        {
            // Помечаем задание как обрабатываемое
            task.Status = ConversionTaskStatus.Converting;
            await _dbContext.SaveChangesAsync(cancellationToken);

            // Скачиваем изображение из S3
            var getObjectResponse = await _s3Client.GetObjectAsync(new
GetObjectRequest
            {
                BucketName = _s3Config.BucketName,
                Key = task.SourceImageId
            }, cancellationToken);

            // Конвертируем изображение
            using var sourceImage =
SKImage.FromEncodedData(getObjectResponse.ResponseStream);
            using var surface = SKSurface.Create(new
SKImageInfo(sourceImage.Width, sourceImage.Height));
            using var canvas = surface.Canvas;
            canvas.DrawImage(sourceImage, 0, 0);

            // Загружаем результат в S3
            var convertedImageId = $"{task.Id}-converted";
```

```
            using var convertedStream = new MemoryStream();
            using var data = surface.Snapshot().Encode(SKEncodedImageFormat.Jpeg,
90);
            data.SaveTo(convertedStream);
            convertedStream.Position = 0;

            await _s3Client.PutObjectAsync(new PutObjectRequest
            {
                BucketName = _s3Config.BucketName,
                Key = convertedImageId,
                InputStream = convertedStream
            }, cancellationToken);

            // Сохраняем идентификатор загруженного изображения и помечаем задание
как выполненное
            task.ConvertedImageId = convertedImageId;
            task.Status = ConversionTaskStatus.Completed;
            await _dbContext.SaveChangesAsync(cancellationToken);

            return new ConvertImageResponse(true);
        }
        catch (Exception ex)
        {
            // В случае ошибки помечаем задание как завершившееся ошибкой
            task.Status = ConversionTaskStatus.Failed;
            await _dbContext.SaveChangesAsync(cancellationToken);

            _logger.LogError(ex, "Error converting image");
            return new ConvertImageResponse(false);
        }
    }
}
```

Проверка статуса задания

Для проверки статуса задания в каталоге `UseCases` создадим каталог `GetTaskStatus` и в нем следующие типы:

- `GetTaskStatusRequest` - запрос на получение статуса задания;
- `GetTaskStatusResponse` - ответ с информацией о статусе задания;
- `IGetTaskStatusHandler` - интерфейс обработчика получения статуса задания;
- `GetTaskStatusHandler` - обработчик получения статуса задания.

Сначала определим тип `GetTaskStatusRequest`:

```
namespace ImageConverter.WebApi.UseCases.GetTaskStatus;

internal sealed record GetTaskStatusRequest(Guid TaskId);
```

Далее определим тип `GetTaskStatusResponse`:

```
namespace ImageConverter.WebApi.UseCases.GetTaskStatus;

internal sealed record GetTaskStatusResponse(ConversionTaskStatus Status);
```

Теперь определим интерфейс `IGetTaskStatusHandler`:

```
namespace ImageConverter.WebApi.UseCases.GetTaskStatus;

internal interface IGetTaskStatusHandler
{
    Task<GetTaskStatusResponse> HandleAsync(GetTaskStatusRequest request,
CancellationToken cancellationToken);
}
```

Далее определим класс `GetTaskStatusHandler`:

```
namespace ImageConverter.WebApi.UseCases.GetTaskStatus;

internal sealed class GetTaskStatusHandler : IGetTaskStatusHandler
{
    private readonly AppDbContext _dbContext;

    public GetTaskStatusHandler(AppDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<GetTaskStatusResponse> HandleAsync(GetTaskStatusRequest
request, CancellationToken cancellationToken)
    {
        var task = await _dbContext.ConversionTasks.FindAsync([request.TaskId],
cancellationToken);
        if (task == null)
        {
            throw new InvalidOperationException($"Task {request.TaskId} not
found");
        }

        return new GetTaskStatusResponse(task.Status);
    }
}
```

Скачивание изображения

Для скачивания изображения в каталоге `UseCases` создадим каталог `DownloadImage` и в нем следующие типы:

- `DownloadImageRequest` - запрос на скачивание изображения;
- `DownloadImageResponse` - ответ с содержимым изображения;
- `IDownloadImageHandler` - интерфейс обработчика скачивания изображения;
- `DownloadImageHandler` - обработчик скачивания изображения.

Сначала определим тип `DownloadImageRequest`:

```
namespace ImageConverter.WebApi.UseCases.DownloadImage;

internal sealed record DownloadImageRequest(Guid TaskId);
```

Далее определим тип `DownloadImageResponse`:

```
namespace ImageConverter.WebApi.UseCases.DownloadImage;

internal sealed record DownloadImageResponse(Stream Content);
```

Теперь определим интерфейс `IDownloadImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.DownloadImage;

internal interface IDownloadImageHandler
{
    Task<DownloadImageResponse> HandleAsync(DownloadImageRequest request,
CancellationToken cancellationToken);
}
```

Далее определим класс `DownloadImageHandler`:

```
namespace ImageConverter.WebApi.UseCases.DownloadImage;

internal sealed class DownloadImageHandler : IDownloadImageHandler
{
    private readonly IAmazonS3 _s3Client;
    private readonly S3Config _s3Config;
    private readonly AppDbContext _dbContext;

    public DownloadImageHandler(IAmazonS3 s3Client, IOptions<S3Config> s3Config,
AppDbContext dbContext)
    {
        _s3Client = s3Client;
        _s3Config = s3Config.Value;
        _dbContext = dbContext;
    }

    public async Task<DownloadImageResponse> HandleAsync(DownloadImageRequest
```

```
request, CancellationToken cancellationToken)
    {
        var task = await _dbContext.ConversionTasks.FindAsync([request.TaskId],
cancellationToken);
        if (task == null)
        {
            throw new InvalidOperationException($"Task {request.TaskId} not
found");
        }

        if (task.Status != ConversionTaskStatus.Completed)
        {
            throw new InvalidOperationException($"Task {request.TaskId} is not
completed");
        }

        if (task.ConvertedImageId == null)
        {
            throw new InvalidOperationException($"Task {request.TaskId} has no
converted image");
        }

        var getObjectResponse = await _s3Client.GetObjectAsync(new
GetObjectRequest
        {
            BucketName = _s3Config.BucketName,
            Key = task.ConvertedImageId
        }, cancellationToken);

        if (getObjectResponse.HttpStatusCode != HttpStatusCode.OK)
        {
            throw new InvalidOperationException($"Failed to download image for
task {request.TaskId}");
        }

        return new DownloadImageResponse(getObjectResponse.ResponseStream);
    }
}
```

Эндпоинты

Теперь добавим эндпоинты для нашего сервиса. Для этого добавим в проект каталог `Endpoints` и в нем файл `ConversionEndpoints.cs`:

```
namespace ImageConverter.WebApi.Endpoints;

internal static class ConversionEndpoints
{
    public static WebApplication MapConversionEndpoints(this WebApplication app)
    {
        app.MapPost("/upload", async (IFormFile file, IUploadImageHandler handler,
```

```
CancellationToken cancellationToken) =>
        {
            using var fileStream = file.OpenReadStream();
            var response = await handler.HandleAsync(new
UploadImageRequest(fileStream), cancellationToken);
            return Results.Ok(response);
        })
        .WithName("UploadImage")
        .WithOpenApi()
        .DisableAntiforgery();

        app.MapGet("/task/{taskId}", async (Guid taskId, IGetTaskStatusHandler
handler, CancellationToken cancellationToken) =>
        {
            var response = await handler.HandleAsync(new
GetTaskStatusRequest(taskId), cancellationToken);
            return Results.Ok(response);
        })
        .WithName("GetTaskStatus")
        .WithOpenApi();

        app.MapGet("/task/{taskId}/converted", async (Guid taskId,
IDownloadImageHandler handler, CancellationToken cancellationToken) =>
        {
            var response = await handler.HandleAsync(new
DownloadImageRequest(taskId), cancellationToken);

            return Results.File(response.Content, "image/jpeg", "converted-
image.jpg");
        })
        .WithName("DownloadImage")
        .WithOpenApi();

        return app;
    }
}
```

Фоновая конвертация

Для запуска конвертации изображений в фоне добавим в проект каталог `BackgroundServices` и в нем файл `ConversionBackgroundService.cs`:

```
namespace ImageConverter.WebApi.BackgroundServices;

internal sealed class ConversionBackgroundService : BackgroundService
{
    private static readonly TimeSpan Delay = TimeSpan.FromSeconds(10);

    private readonly ILogger<ConversionBackgroundService> _logger;
    private readonly IServiceScopeFactory _serviceScopeFactory;
```

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

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("ConversionBackgroundService is starting.");

        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                var result = await HandleConversionAsync(stoppingToken);

                switch (result)
                {
                    case HandlerResult.NoTasks:
                    case HandlerResult.Error:
                        await Task.Delay(Delay, stoppingToken);
                        break;
                    case HandlerResult.Converted:
                        break;
                }
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error in ConversionBackgroundService");
                await Task.Delay(Delay, stoppingToken);
            }
        }
    }

    private async Task<HandlerResult> HandleConversionAsync(CancellationToken
cancellationToken)
    {
        using var scope = _serviceScopeFactory.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

        var task = await dbContext.ConversionTasks
            .OrderBy(t => t.CreatedAt)
            .FirstOrDefaultAsync(t => t.Status == ConversionTaskStatus.Pending,
cancellationToken);

        if (task == null)
        {
            return HandlerResult.NoTasks;
        }

        var handler =
scope.ServiceProvider.GetRequiredService<IConvertImageHandler>();
```

```
        try
        {
            var response = await handler.HandleAsync(new
ConvertImageRequest(task.Id), cancellationToken);

            return response.Success ? HandlerResult.Converted :
HandlerResult.Error;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in HandleConversionAsync");
            return HandlerResult.Error;
        }
    }

    private enum HandlerResult
    {
        NoTasks,
        Converted,
        Error
    }
}
```

Как видим, внутри данного сервиса мы периодически проверяем наличие заданий на конвертацию изображений и выполняем их при помощи написанного ранее обработчика. Так как мы вынесли логику конвертации в отдельный обработчик, то ничто не мешает нам в дальнейшем для запуска использовать, например, консьюмер Kafka.

Собираем все вместе

Теперь, когда у нас есть все необходимые компоненты, мы можем собрать все вместе.

Сначала зарегистрируем все необходимые сервисы в DI-контейнере. Для этого в файле `Program.cs` перед кодом `var app = builder.Build();` добавим следующий код:

```
// Регистрация контекста базы данных
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString("Default")));

// Регистрация сервиса для выполнения миграций
builder.Services.AddHostedService<MigrationRunner>();

// Регистрация обработчиков запросов
builder.Services.AddScoped<IConvertImageHandler, ConvertImageHandler>();
builder.Services.AddScoped<IUploadImageHandler, UploadImageHandler>();
builder.Services.AddScoped<IDownloadImageHandler, DownloadImageHandler>();
builder.Services.AddScoped<IGetTaskStatusHandler, GetTaskStatusHandler>();

// Регистрация фонового сервиса для конвертации изображений
builder.Services.AddHostedService<ConversionBackgroundService>();
```

```
// Регистрация конфигурации для S3
builder.Services
    .AddOptions<S3Config>()
    .Bind(builder.Configuration.GetSection("S3"))
    .ValidateDataAnnotations();

// Регистрация клиента для S3
builder.Services.AddSingleton<IAmazonS3>(sp =>
{
    var configuration = sp.GetRequiredService<IOptions<S3Config>>().Value;

    return new AmazonS3Client(
        configuration.AccessKey,
        configuration.SecretKey,
        new AmazonS3Config
        {
            ServiceURL = configuration.ServiceUrl,
            ForcePathStyle = true,
        }
    );
});
```

Теперь добавим в файл `appsettings.json` конфигурацию для S3 и базы данных:

```
{
  "S3": {
    "AccessKey": "minioadmin",
    "SecretKey": "minioadmin",
    "ServiceUrl": "http://localhost:9000",
    "BucketName": "image-converter"
  },
  "ConnectionStrings": {
    "Default": "Host=localhost;Database=image-
converter;Username=postgres;Password=postgres"
  }
}
```

Добавление в Docker Compose

Теперь нужно добавить наше приложение в Docker Compose. Но перед этим нам необходим Dockerfile для нашего приложения.

В каталоге с решением создадим файл `Dockerfile`:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

COPY ["ImageConverter.WebApi/ImageConverter.WebApi.csproj",
"ImageConverter.WebApi/"]
```

```
RUN dotnet restore "ImageConverter.WebApi/ImageConverter.WebApi.csproj"

COPY . .

WORKDIR "/src/ImageConverter.WebApi"

RUN dotnet publish -c Release -o /out

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS publish
WORKDIR /app
COPY --from=build /out .
ENTRYPOINT ["dotnet", "ImageConverter.WebApi.dll"]
```

Теперь добавим наше приложение в Docker Compose.

В файл `docker-compose.yml` добавим следующий код:

```
services:
  # ... другие сервисы
  image-converter:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "5000:8080"
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__Default=Host=postgres;Database=image-
converter;Username=postgres;Password=postgres
      - S3__AccessKey=minioadmin
      - S3__SecretKey=minioadmin
      - S3__ServiceUrl=http://minio:9000
      - S3__BucketName=image-converter
    depends_on:
      - postgres
      - minio
      - createbuckets
```

Теперь запустим наше приложение:

```
docker compose up -d
```

Теперь наше приложение будет доступно по адресу `http://localhost:5000`.