.NET Concepts: Services

Release

Oct 28, 2017

Contents:

1 Getting Started **3**
- 1.1 Create Service . . . 3
- 1.2 Create Bootstrap . . . 3
- 1.3 Run the service . . . 4

2 Service Host **5**
- 2.1 Hosting ASP.NET Core . . . 5

3 RabbitMQ **7**

.NET Concepts: Services are concept libraries made to inspire a better structure for C# services.

CHAPTER 1

Getting Started

Create Service

A service is defined by implementing a class derived from `Service`. The method `StartAsync` will be called when the service is started, and can be considered as the entry point to the service. Optionally, `StopAsync` can be overridden to implement clean up activities, like disposing services.

```
public class TimeService : Service
{
  private readonly IWorldClock _clock;
  private Timer _timer;

  public TimeService(IWorldClock clock)
  {
    _clock = clock;
  }

  public override async Task StartAsync(CancellationToken ct =␣
↪default(CancellationToken))
  {
    _timer = new Timer(time =>
    {
      Log.Information("It is {timeOfDay}, and all is well", _clock.GetTime());
    }, null, TimeSpan.Zero, TimeSpan.FromSeconds(10));
  }
}
```

Create Bootstrap

The `IServiceBootstrap` is responsible for configuring the applicatoin logger and wire-up the dependency injection container. It is not primed to any specific frameworks, as the interface only contains hooks. For convinience, there are implementations that wire up different populare libraries.

The `OpinionatedServiceBootstrap` configures a Serilog logger and creates an Autofac container to register services in.

```
public class TimeBootstrap : OpinionatedServiceBootstrap<TimeService>
{
  public override ServiceMetadata CreateMetadata()
  {
    return new ServiceMetadata
    {
      Type = typeof(TimeService),
      Name = nameof(TimeService),
      Description = "Tells the time"
    };
  }

  protected override void RegisterDependencies(ContainerBuilder builder)
  {
    builder
      .RegisterType<WorldClock>()
      .AsImplementedInterfaces();
    builder
      .RegisterType<TimeService>()
      .AsSelf();
  }
}
```

Run the service

The service can be run in a few different ways. The most straight forward option is to use the `ConsoleRuner`

```
public class Program
{
    public static void Main(string[] args)
    {
        MainAsync(args).GetAwaiter().GetResult();
    }

    public static async Task MainAsync(string[] args)
    {
        await ConsoleRunner.StartAsync(new TimeBootstrap());
    }
}
```

The console runner is a great option for running services on .NET Core in Docker containers. On a Windows system, the `TopshelfRunner` can be used to run the service as an actual Windows Service

```
TopshelfRunner.Start(new TimeBootstrap());
```

There are more sophisticated ways to run a service, that allows hybrid services that runs an ASP.NET Core API as well as a traditional service. This is achieved by using the `ServiceHost` and related classes.

CHAPTER 2

Service Host

The `ServiceHost` is heavely inspired by the `WebHost` classes, used to configure ASP.NET Core applications.

A service host is created by defining a `ServiceHostBuilder`, configuring it and finally building the host.

```
var serviceHost = new ServiceHostBuilder(new TimeBootstrap())
    .UseConsoleHost()
    .Build();

await serviceHost.RunAsync();
```

Hosting ASP.NET Core

The package `Concept.Service.AspNetCore` contains classes that makes it possible to host an ASP.NET application together with the service. This can be great in a microservice architecture where each service exposes an API as well as underlying, event based business logic.

Getting up and running is fairly easy. Make sure that the bootstrap inherits from `AspNetCoreBootstrap`. This is an extended bootstrap that, in addition to normal bootstrapping, contains methods similar to the ones in the `Startup` class.

```
public class FooBootstrap : AspNetCoreBootstrap<FooService>
{
  public override void ConfigureServices(IServiceCollection services)
  {
    services
      .AddSingleton<FooService>()
      .AddLogging()
      .AddMvc();
  }

  public override void ConfigureAppConfiguration(IConfigurationBuilder configuration)
  {
    configuration.AddJsonFile("appsettings.json");
```

```
  }

  public override void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment())
      app.UseDeveloperExceptionPage();

    app.UseMvc();
  }
}
```

With the updated bootstrapper, the service host builder can define multiple hosts

```
public static async Task MainAsync(string[] args)
{
  var host = new ServiceHostBuilder(new OpinionatedFooBootstrap())
    .UseConsoleHost()
    .UseWebHost()
    .Build();

  await host.RunAsync();
}
```

CHAPTER 3

RabbitMQ

RabbitMQ is a populare message broker for distributed systems. The package `Concept.Service.RabbitMq` contains `RabbitMqService` that has methods for subscribing and publishing messages. It uses RawRabbit under the hood.

```
public class FooService : RabbitMqService
{
  public FooService(IBusClient busClient) : base(busClient) { }

  public override async Task StartAsync(CancellationToken ct =␣
↪default(CancellationToken))
  {
    // Method in base class
    await SubscribeAsync<PerformFoo>(HandleFooAsync, ct: ct);
  }

  private async Task HandleFooAsync(PerformFoo message, ConceptContext context)
  {
    /* Handle message */
    // Method in base class
    await PublishAsync(new FooPerformed {Success = true});
  }
}
```