Если суммировать, то можно выделить следующие ключевые отличия ASP.NET Core от предыдущих версий ASP.NET:

* Новый легковесный и модульный конвейер HTTP-запросов
* Возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса
* Использование платформы .NET Core и ее функциональности
* Распространение пакетов платформы через NuGet
* Интегрированная поддержка для создания и использования пакетов NuGet
* Единый стек веб-разработки, сочетающий Web UI и Web API
* Конфигурация для упрощенного использования в облаке
* Встроенная поддержка для внедрения зависимостей
* Расширяемость
* Кроссплатформенность: возможность разработки и развертывания приложений ASP.NET на Windows, Mac и Linux
* Развитие как open source, открытость к изменениям

Для обработки запросов теперь используется новый конвейер HTTP, который основан на компонентах Katana и спецификации OWIN. А его модульность позволяет легко добавить свои собственные компоненты.

var host = new WebHostBuilder()

        .UseKestrel()               // настраиваем веб-сервер Kestrel

        .UseContentRoot(Directory.GetCurrentDirectory())    // настраиваем корневой каталог приложения

        .UseIISIntegration()        // обеспечиваем интеграцию с IIS

        .UseStartup<Startup>()    // устанавливаем главный файл приложения

        .Build();                   // создаем хост

    host.Run();                     // запускаем приложение

**StartUp]**

public void ConfigureServices(IServiceCollection services)//Optional either //like constructor

        {

        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)

        {

            if (env.IsDevelopment())

            {

                app.UseDeveloperExceptionPage();

            }

            app.Run(async (context) =>

            {

                await context.Response.WriteAsync("Hello World!");

            });

        }

    }

Метод Configure выполняется один раз при создании объекта класса Startup, и компоненты middleware создаются один раз и живут в течение всего жизненного цикла приложения:

public void Configure(IApplicationBuilder app)

    {

        int x = 2;

        app.Run(async (context) =>

        {

            x = x \* 2;  //  2 \* 2 = 4 //Next request = 8

            await context.Response.WriteAsync($"Result: {x}");

        });

    }

Перегрузка метода Use, которая в качестве параметров принимает контекст запроса - объект HttpContext и делегат Func<Task>, который представляет собой ссылку на следующий в конвейере компонент middleware. Делится на две части (до next.Invoke() и после next.Invoke().)

Run не вызывают никакие другие компоненты и дальше обработку запроса не передают.

Метод Map (и методы расширения MapXXX()) применяется для сопоставления пути запроса с определeнным делегатом, который будет обрабатывать запрос по этому пути. Например: app.Map("/index", Index);

private static void Index(IApplicationBuilder app)

{

    app.Run(async context =>

    {

        await context.Response.WriteAsync("Index");

    });

}

Вложенные методы Map:

app.Map("/home", home =>

    {

        home.Map("/index", Index);

        home.Map("/about", About);

    });

Метод MapWhen

app.MapWhen(context => {

        return context.Request.Query.ContainsKey("id") &&

                context.Request.Query["id"] == "5";

    }, HandleId);

Class component

public class TokenMiddleware

{

    private readonly RequestDelegate \_next;

    public TokenMiddleware(RequestDelegate next)

    {

        this.\_next = next;

    }

    public async Task InvokeAsync(HttpContext context)

    {        if (token!="12345678")

        {//some code        }

        else

        {

            await \_next.Invoke(context);

        }

    }

}

!!!!!!!!! In StartUp > Configure > app.UseMiddleware<TokenMiddleware>() or use extension method like UseToken(this IApplicationBuilder builder).If you want to pass parameter you can use constructor.

**static**

WebHost.CreateDefaultBuilder(args)

            .UseStartup<Startup>()          // установка класса Startup как стартового

            .UseWebRoot("static")   // установка папки static

*http://localhost:55234 /index.html – from static*

**UseDefaultFiles() ->**  [*http://localhost/*](http://localhost/) *->* default.htm, default.html,index.htm, index.html or DefaultFilesOptions options = new DefaultFilesOptions();

        options.DefaultFileNames.Clear(); // удаляем имена файлов по умолчанию

        options.DefaultFileNames.Add("hello.html"); // добавляем новое имя файла

        app.UseDefaultFiles(options); // установка параметров

**app.UseDirectoryBrowser() ->**  позволяет пользователям просматривать содержимое каталогов на сайте

app.UseDirectoryBrowser(new DirectoryBrowserOptions()

    {

        FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),

        RequestPath = new PathString("/pages")

    });

OWIN (Open Web Interface for .NET), которая позволяет отвязать веб-приложение от конкретного веб-сервера и по сути создать самохостирующееся приложение. OWIN определеляет механизм использования компонентов middleware для обработки конкретных запросов и отправки ответа.

В соответствии со спецификацией OWIN компоненты middleware OWIN должны принимать объект IDictionary<string, object>, который представляет набор параметров, связанных с запросом и возвращаемым ответом.

начиная с версии ASP.NET Core 2.0 мы не можем передавать scoped-сервисы в конструктор singleton-объектов: Напр : на момент создания объекта TimerMiddleware scoped-сервис TimeService еще не установлен, соответственно он использоваться не может. А без создания объекта TimeService нельзя создать объект TimerMiddleware.

Аналогичная ситуация может возникнуть, если TimeService добавляется как Transient, а сервис ITimer определен как Scoped: В этом случае для создания объекта TimeService надо получить сервис ITimer, но на момент вызова конструктора TimerMiddleware сервис ITimer еще неопределен.

Configuraation

Создание конфигурации вовлекает три компонента: **IConfigurationSource** (определяет источник конфигурации- его метод Build(IconfigurationBuilder) returns provider),**ConfigurationProvider** (сам провайдер конфигурации – его метод Load() fills his property Data) и некий класс, который добавляет метод расширения к объекту IConfiguration(his method adds to builder source and returns this builder).

Начиная с версии 2.0, необязательно явным образом подключать конфигурацию в проект. Чтобы получить конфигурацию из файла appsettings.json, достаточно передать в конструктор класса объект IConfiguration, который и будет содержать все настройки. Более того конфигурация, составленная из файла appsettings.json, доступна по всему приложению, и мы можем ее получить через механизм Dependency Injection(просто инъекция IConfiguration например в коструктор middleware) Но: это работает только с файлом appsettings.json – в других случаях для этого нужно -🡪 services.AddTransient<IConfiguration>(provider => AppConfiguration);

**Передача конфигурации через IOptions(инъекция в конструктор** IOptions<Person>**) :**

в ConfigureServices указываем:  // Настройка параметров и DI

            services.AddOptions();

            // создание объекта Person по ключам из конфигурации

            services.Configure<Person>(AppConfiguration);

также при необходимости мы можем переопределить настройки: services.Configure<Person>(opt =>

    {

        opt.Age = 22;

    });

Также можно передавать отдельные секции конфигурации services.AddOptions();

    services.Configure<Person>(AppConfiguration);

    services.Configure<Company>(AppConfiguration.GetSection("company"));

Запуск в процессах отличных от IIS

ASP.NET Core предоставляет возможность запускать приложение без IIS в рамках собственного процесса с помощью двух дополнительных http-серверов, которые идут вместе с ASP.NET Core:

* Microsoft.AspNetCore.Server.HttpSys (или просто WebListener) (в предыдущих версиях ASP.NET Core назывался WebListener)
* Microsoft.AspNetCore.Server.Kestrel (или просто Kestrel)

HTTP.sys работает только на платформе Windows, а Kestrel является кроссплатформенным.

Хостирование приложений ASP.NET Core на IIS происходит с помощью нативного модуля IIS под названием **AspNetCoreModule**, который сконфигурирован таким образом, чтобы перенаправлять запросы на веб-сервер Kestrel. Этот модуль управляет запуском внешнего процесса dotnet.exe, в рамках которого хостируется приложение, и перенаправляет все запросы от IIS к этому хостирующему процессу.

Веб-сервер Kestrel получает запрос и передает его в виде объекта HttpContext в конвейер middleware ASP.NET Core. Конвейер middleware в приложении обрабатывает запрос и возвращает IIS результат обработки, который затем посылается HTTP-клиенту (например, веб-бразеру). IIS+ это кеширование статических файлов. Kestrel представляет кроссплатформенный веб-сервер, основанный на кросплатформенной библиотеке асинхронного ввода/вывода libuv. Kestrel(кроссплатформенный веб-сервер) использует сокеты и полностью состоит из управляемого кода. Kestrel развивается как opensource-проект, и при необходимости на гитхабе можно посмотреть его [исходный код](https://github.com/aspnet/KestrelHttpServer).

При развертывании на Windows Kestrel может применять IIS в качестве прокси-сервера, а при развертывании на Linux как прокси-серверы могут использоваться Apache и Nginx. Но также Kestrel может работать самостоятельно внтури своего процесса без IIS.

Прокси-серверы(IIS, Apache, Nginx) позволяет скрыть приложения, если они не должны быть доступны напрямую. Кроме того, веб-серверы позволяет управлять нагрузкой ко всем приложениям, и предоставляют другие функции по управлению приложениями.

HTTP.sys(UseHttpSys() вместо UseKestrel()) представляет HTTP-сервер для ASP.NET Core, который работает только в ОС Windows. Ранне данный сервер назывался WebListener. Он запускается поверх драйвера ядра **Http.Sys**. Весь функционал сервера сосредоточен в пакете **Microsoft.AspNetCore.Server.HttpSys**.

ASP.NET Core можно развертывать в виде обычной службы Windows без каких-либо веб-серверов, в частности, IIS. Прежде всего, нам надо добавить в проект через Nuget пакет**Microsoft.AspNetCore.Hosting.WindowsServices**. Чтобы запустить приложение в виде службы у объекта IWebHost вызывается метод host.**RunAsService()**.Теперь нам надо опубликовать приложение в файловой системе: dotnet publish --configuration Release --runtime win10-x64 --output c:\myapp

**Далее создаем службу:** sc create НАЗВАНИЕ\_СЛУЖБЫ binPath= "ПУТЬ К ИСПОЛНЯЕМОМУ ФАЙЛУ EXE"

**Далее запускаем службу:** sc start MyAspService.

**Npm vs Bower**

Возможно, для зависимости B требуется другая версия зависимости A, чем для зависимости C. npm устанавливает обе версии этой зависимости, поэтому она все равно будет работать, но Bower создаст вам конфликт, потому что не любит дублирование (поскольку загрузка одного и того же ресурса на веб-странице очень неэффективно и дорого, а также может привести к серьезным ошибкам). Вам нужно будет вручную выбрать, какую версию вы хотите установить. Это может привести к тому, что одна из зависимостей будет нарушена, но это нужно исправить в любом случае.

**Grunt vs Gulp**

module.exports = function (grunt) {

    grunt.initConfig({

        concat: {

            build: {

                src: ['wwwroot/js/\*'],

                dest: 'wwwroot/js/app.js'

            }

        },

        uglify: {

            build: {

                files: {

                    'wwwroot/bundles/jquery.js': ['wwwroot/lib/jquery/dist/jquery.js'],

                    'wwwroot/bundles/jqueryval.js': ['wwwroot/lib/jquery-validation/dist/jquery.validate.js', 'lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js'],

                    'wwwroot/bundles/bootstrap.js': ['wwwroot/lib/bootstrap/dist/js/bootstrap.min.js'],

                    'wwwroot/bundles/app.js': ['wwwroot/js/app.js']

                }

            }

        }

grunt.loadNpmTasks('grunt-contrib-clean');

    grunt.loadNpmTasks('grunt-contrib-copy');

    grunt.loadNpmTasks('grunt-contrib-cssmin');

    grunt.loadNpmTasks('grunt-contrib-concat');

    grunt.loadNpmTasks('grunt-contrib-uglify');

    grunt.registerTask('build', ['clean', 'copy', 'concat', 'uglify', 'cssmin']);

"use strict";

var gulp = require("gulp"),

    rimraf = require("rimraf"),

    concat = require("gulp-concat"),

    cssmin = require("gulp-cssmin"),

    uglify = require("gulp-uglify");

var paths = {

    webroot: "./wwwroot/"

};

paths.js = paths.webroot + "js/\*\*/\*.js";

paths.minJs = paths.webroot + "js/\*\*/\*.min.js";

paths.css = paths.webroot + "css/\*\*/\*.css";

paths.minCss = paths.webroot + "css/\*\*/\*.min.css";

paths.concatJsDest = paths.webroot + "js/site.min.js";

paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {

    rimraf(paths.concatJsDest, cb);

});

gulp.task("clean:css", function (cb) {

    rimraf(paths.concatCssDest, cb);

});

gulp.task("clean", ["clean:js", "clean:css"]);

***Grunt – все последовательно***

***Gulp – параллельно***