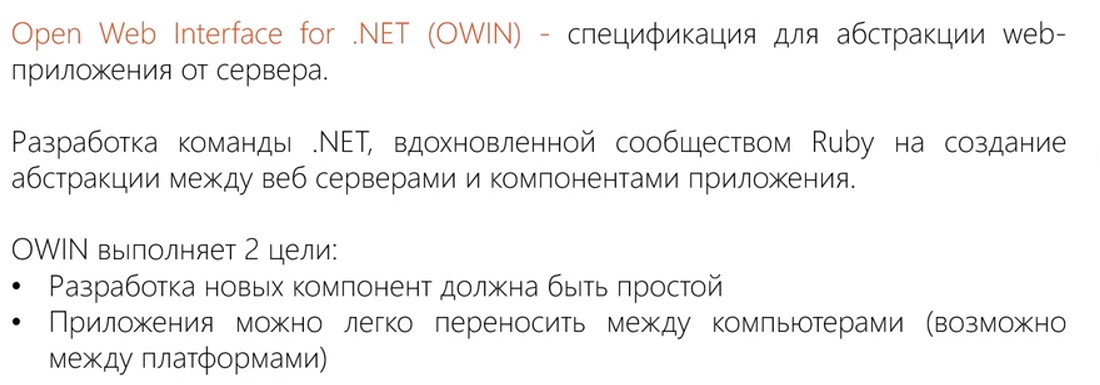
# Что входит в ASP.NET Core Empty

* **Connected Services**: подключенные сервисы из Azure
* **Dependencies**: все добавленные в проект пакеты и библиотеки, иначе говоря зависимости
* **Properties**: узел, который содержит некоторые настройки проекта. В частности, в файле launchSettings.json описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.
* **appsettings.json**: файл конфигурации проекта в формате json
* **Program.cs**: главный файл приложения, с которого и начинается его выполнение. Код этого файла настривает и запускает веб-хост, в рамках которого разворачивается приложение
* **Startup.cs**: файл, который определяет класс Startup и который содержит логику обработки входящих запросов

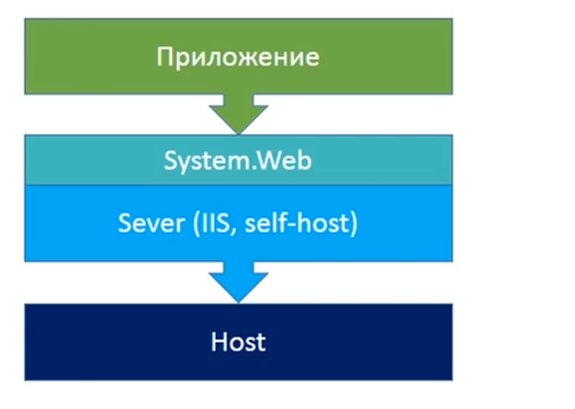
# OWIN



Это сводка правил

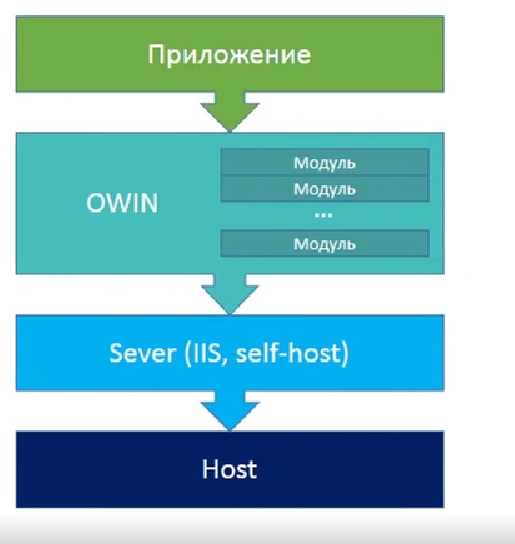
Идея – сделать обработку запросов конвеерной

**Была проблема**, сразу после запуска, поднятие приложения, выделение потоков, обработака запросов и тд лежит на сервере, **вся бизнес логика лежит на серверной части**, и разделения между приложением и сервером не было, нельзя было взять приложение и перенести его на другой сервер



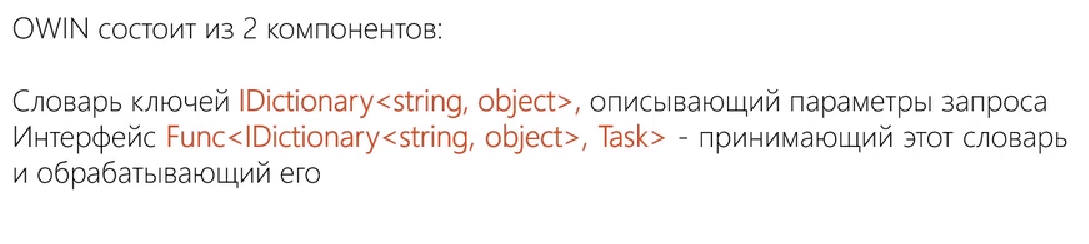
Microsoft начали исправлять, сделали MVC API SIGNALR, так отделились сервер и приложение

Получилась прослойка, называемая OWIN, в котором был весь технический функционал, такой как авторизация, обработка запросов, ошибок, логирование и тд



Спецификация:

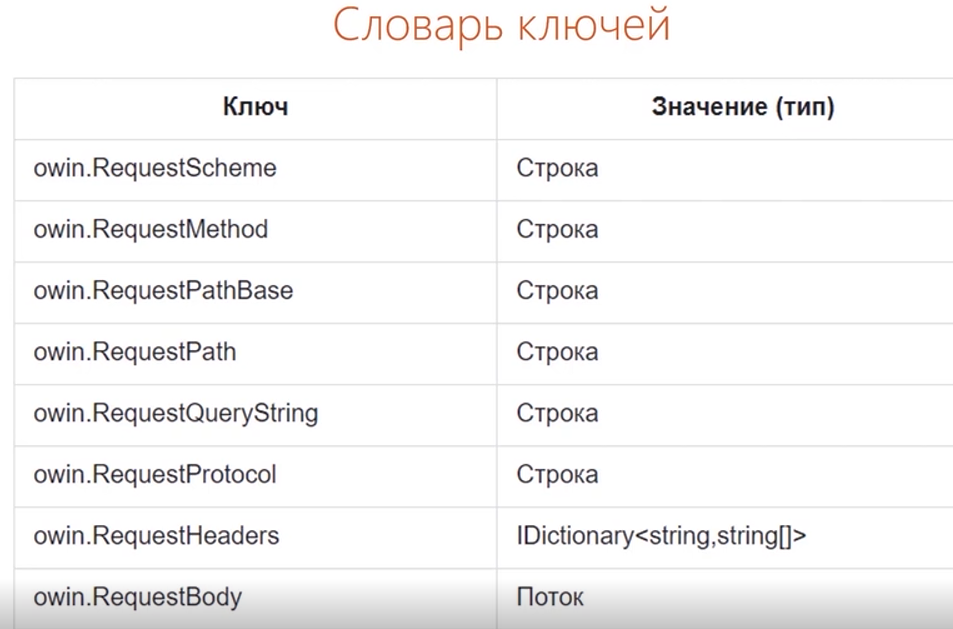
OWIN -абстракция сервера, то есть как модуль должен выглядеть чтобы быть понятным серверу



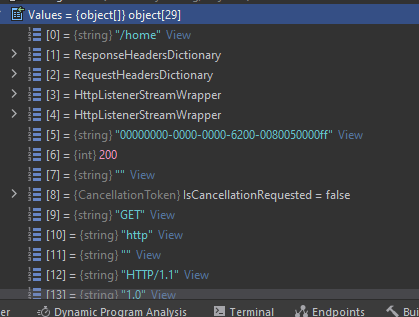
Словарь – параметры запроса

Функция – обработка запроса

Словарь выглядит так:



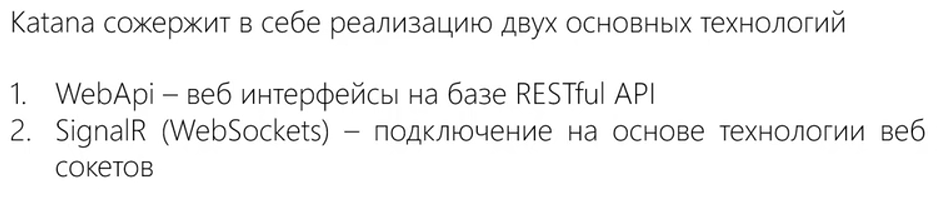
Словарь значений



Окей, как это юзать, чтобы не писать все юзаем Katana framework

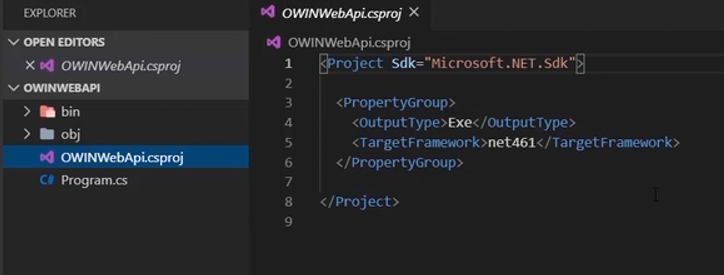
## Katana

Презентация owin for c#



Реализуем свой обработчик

Реализуем свой обработчик *framework* чтобы реализовать API чтобы пониамать все



Хдесь будут зависимости

Добавляем с Nuget’a:



Еще нужна либа для развертывания OWIN приложения



Теперь начнем реализацию, добавим главный файл – startup.cs

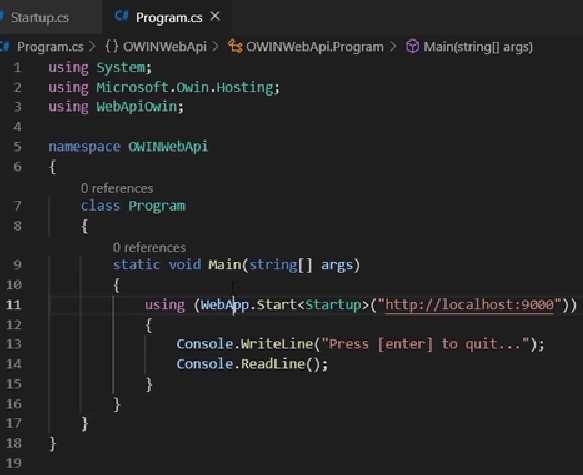


## Сделаем свой webApi в консольном приложении

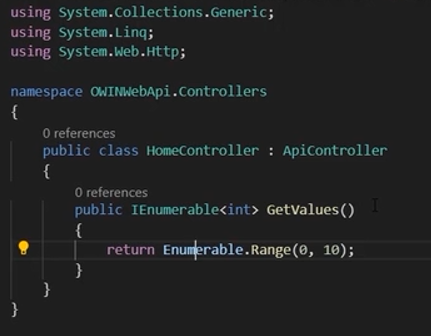
Катана ищет класс страртап и метод Configuration, это сборщик, куда мы будем все подключать,

В примере Здесь мы создаем конфиг, определяем петь для контроллеров и подключаем web api

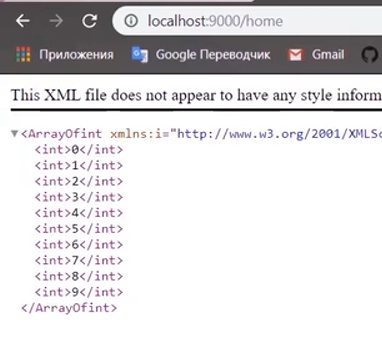
Теперь в мэйне юзаем наш класс через OWIN,



Создадим контроллер

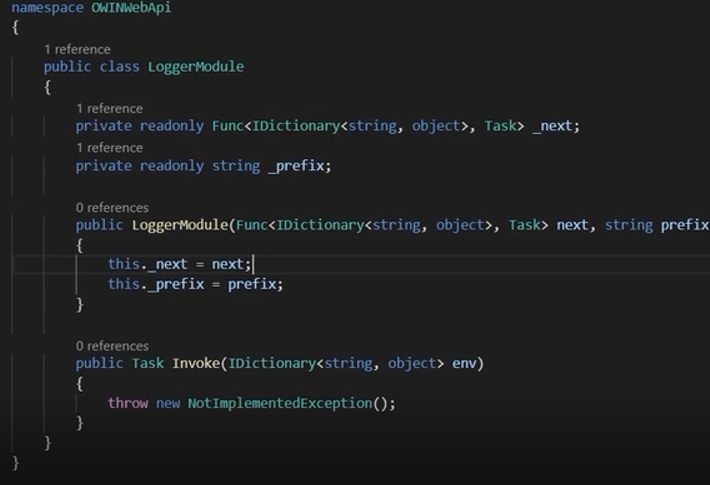


Тперь видно

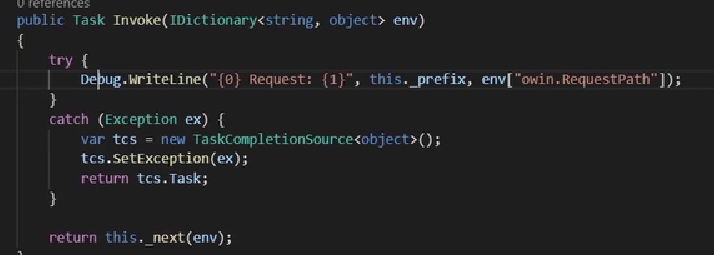


Приложение перенаправило нас на контроллер

**Давайте сделаем логгер в соответствии с OWIN**

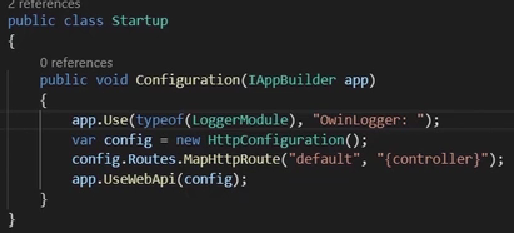


В Invoke будет обработчик запроса, делаем



В конце вызываем следующую таску переданную извне

Пропишем в стартапе



Если перейти localhost:4444/home

In console: /home

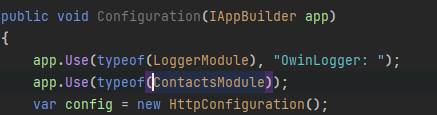
Чтобы отвечал response’ами – ниже

## Controller

Простейший контроллер, который возращает инты на

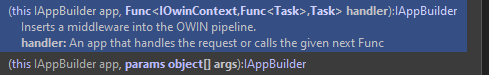
# Как это все вместе работает

Есть у нас конфиг функция

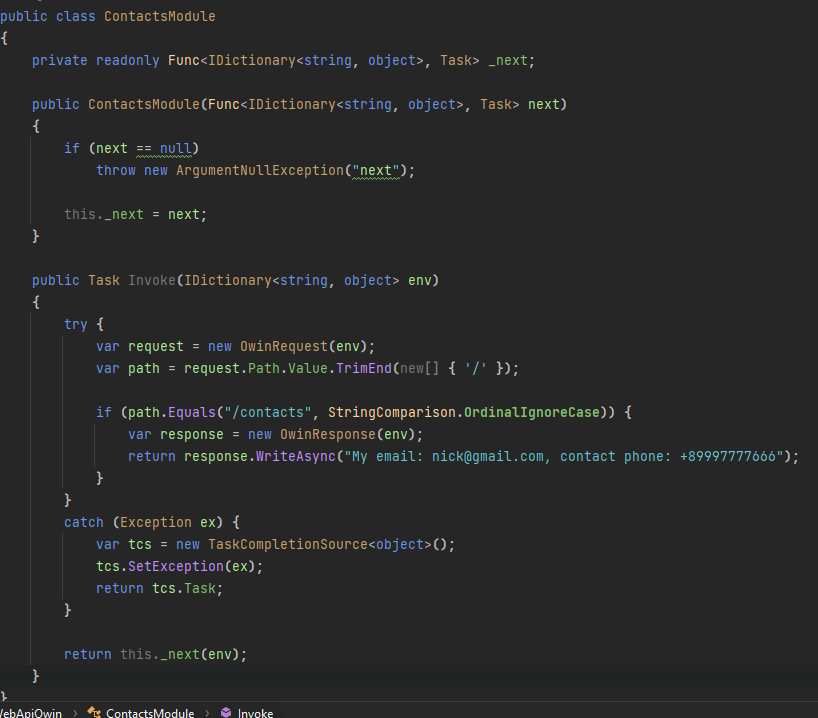


Есть метод use, мы можем юзать его так, а можем передавать next еще

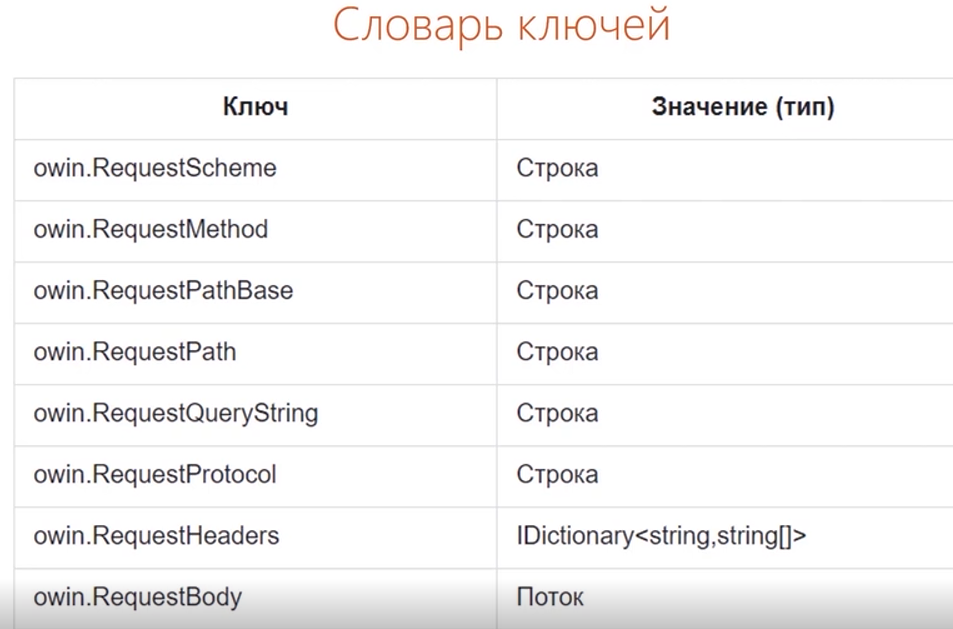
Теперь отсюда метод Use пихает туда dictionary с инфой о запросе, и мы определяем функцию- обработчик запроса, негласно это Invoke/InvokeAsync



ContactsModule:



Use запускает конструктор с инфой о запросе и методами всякими:



Потом запускает Invoke, там мы возращаем таску, которую надо выполнить

ВСЕ, ну тут у нас одни get’ы так что все просто он просто возращает то что нужно

/home and /contacts works

# Web api

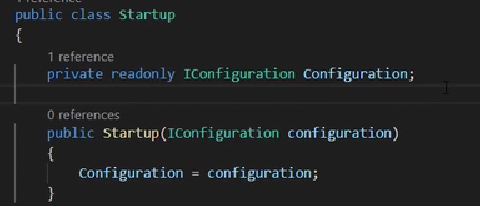
В startapp’e определяем Configure and ConfigureServices

ConfigureServices

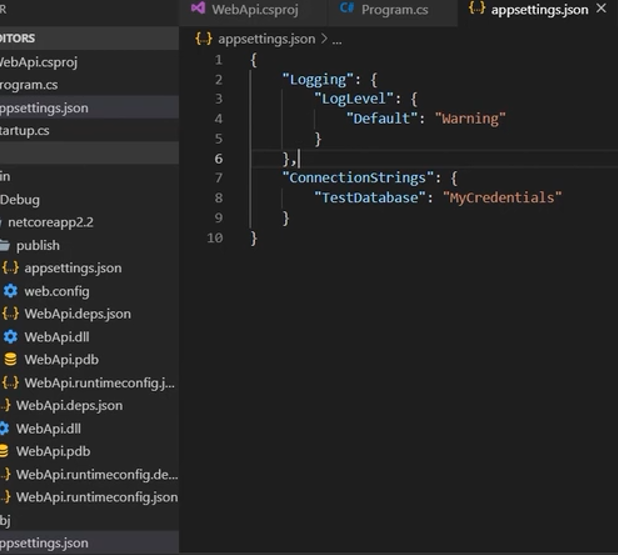
–все манипуляции с сервисами приложения, сервисы приложения это модули которые связаны друг с другом через **DependencyInjection**

## Конфигурация app

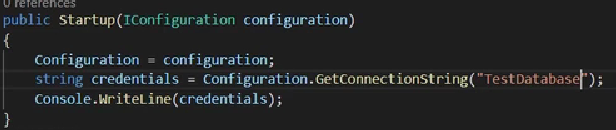
Первый сервис который есть в приложении это сервис конфигурации приложения, чтобы его включить его надо сделать конструктор с параметрами



Теперь механизм DependencyInjection сам будет смотреть на конструктор, смотреть параметры и если параметр – сервис то он будет вставлять его при создании нашего класса

**Чтож давайте попробуем создать свой json чтобы создать файл конфигурации **

Теперь мы можем получить наш connectionstring:



17:00 закончил

# Host.CreateDefaultBuilder(args)

* Устанавливает корневой каталог (для этого используется свойство Directory.GetCurrentDirectory). Корневой каталог представляет папку, где будет производиться поиск различного содержимого, например, представлений.
* Устанавливает конфигурацию хоста. Для этого загружаются переменные среды с префиксом "DOTNET\_" и аргументы командной строки
* Устанавливает конфигурацию приложения. Для этого загружается содержимое из файлов appsettings.json и appsettings.{Environment}.json, а также переменные среды и аргументы командной строки. Если приложение в статусе разработки, то также используются данные Secret Manager (менеджера секретов), который позволяет сохранить конфиденциальные данные, используемые при разработке.
* Добавляет провайдеры логирования
* Если проект в статусе разработки, то также обеспечивает валидацию сервисов
* Можно настроить корневой каталог приложения
* Конфиг узла
* Взять настройки арр.джейсона

# ConfigureWebHostDefaults().

* Загружает конфигурацию из переменных среды с префиксом "ASPNETCORE\_"
* Запускает и настраивает веб-сервер Kestrel, в рамках которого будет разворачиваться приложение
* Добавляет компонент Host Filtering, который позволяет настраивать адреса для веб-сервера Kestrel
* Если переменная окружения ASPNETCORE\_FORWARDEDHEADERS\_ENABLED равна true, добавляет компонент Forwarded Headers, который позволяет считывать из запроса заголовки "X-Forwarded-"
* Если для работы приложения требуется IIS, то данный метод также обеспечивает интеграцию с IIS

## useStartup

А помощью последовательного вызова цепочки методов у объекта IWebHostBuilder производится инициализация веб-сервера для развертывания веб-приложения. В частности, в данном случае у IWebHostBuilder вызывается метод **UseStartup()**:

|  |  |
| --- | --- |
| 1 | webBuilder.UseStartup<Startup>() |

Этим вызовом устанавливается стартовый класс приложения - класс Startup, с которого и будет начинаться обработка входящих запросов.

# Startup class

* конфигурацию приложения,
* настраивает сервисы, которые приложение будет использовать,
* устанавливает компоненты для обработки запроса или middleware.

Класс Startup должен определять метод Configure(), и также опционально в Startup можно определить конструктор класса и метод ConfigureServices().

При запуске приложения сначала срабатывает *конструктор*, затем метод ConfigureServices() и в конце метод Configure(). Эти методы вызываются средой выполнения ASP.NET.

## ConfigureServices

Необязательный метод ConfigureServices() регистрирует сервисы, которые используются приложением. В качестве параметра он принимает объект **IServiceCollection**, который и представляет коллекцию сервисов в приложении. С помощью методов расширений этого объекта производится конфигурация приложения для использования сервисов. Все методы имеют форму Add[название\_сервиса].

К примеру, в проекте по типу **Web Application (Model-View-Controller)** данный метод имеет следующее определение

|  |  |
| --- | --- |
| 1  2  3  4 | public void ConfigureServices(IServiceCollection services)  {      services.AddControllersWithViews();  } |

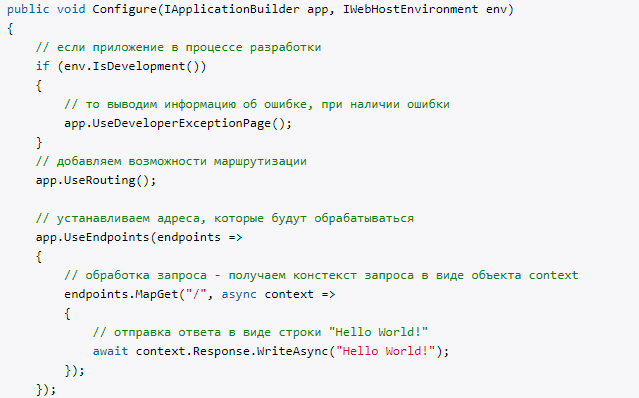
Метод services.AddControllersWithViews(); добавляет в коллекцию сервисов сервисы, которые необходимы для работы контроллеров MVC.

## Configure

Метод Configure устанавливает, как приложение будет обрабатывать запрос

Для установки компонентов, которые обрабатывают запрос, используются методы объекта **IApplicationBuilder**. Объект IApplicationBuilder является обязательным параметром для метода Configure.

Кроме того, метод нередко принимает еще один необязательный параметр - объект **IWebHostEnvironment**, который позволяет получить информацию о среде, в которой запускается приложение, и взаимодействовать с ней.



* if (env.IsDevelopment()) проверяет, находится ли приложение в состоянии/статусе разработки. Что это значит? Для проекта можно указать, например, через настройки, что он находится в состоянии разработки. Вообще условно есть три состояния или стадии проекта: в состоянии разработки (**Development**), в состоянии подготовки к развертыванию (**Staging**) и в состоянии полноценного использования (**Production**), когда он уже развернут на каком-нибудь сервере, и пользователи могут к нему обращаться. По умолчанию Visual Studio устанавливает для проекта состояние разработки. И данное выражение как раз проверяет состояние.
* app.UseDeveloperExceptionPage();

который выводит подробные сообщения об ошибках. Подобные сообщения нежелательны и могут раскрывать некоторые чувствительные данные, когда приложение уже развернуто на сервере и с ним могут работать пользователи, поэтому они подобные сообщения по умолчанию выводятся только в состоянии разработки.

* Вызов **app.UseRouting()** добавляет некоторые возможности маршрутизации, благодаря чему приложение может соотносить запросы с определенными маршрутами.
* Далее идет вызов app.UseEndpoints(endpoints =>, который позволяет определить маршруты, которые будут обрабатываться приложением.
* Цепочка вызовов завершается выражением

endpoints.MapGet("/", async context =>

{

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

});

Это выражение указывает, что для всех запросах по маршруту "/" (то есть к корню веб-приложения) в ответ будет отправляться строка "Hello World!".

## Startup ctor

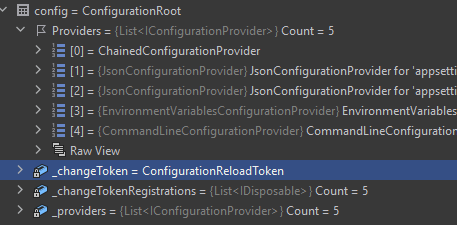
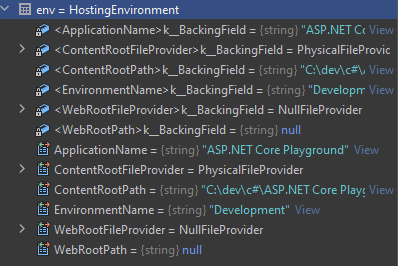
Конструктор Startup

Конструктор является необязательной частью класса Startup. В конструкторе, как правило, производится начальная конфигурация приложения.

Можно создать конструктор без параметров, а можно в качестве параметров передать сервисы **IWebHostEnvironment** (передает информацию о среде, в которой запускается приложение) и **IConfiguration** (передает конфигурацию приложения), которые доступны для приложения по умолчанию.

К примеру, можно получить доступный для приложения по умолчанию сервис IWebHostEnvironment, сохранить его в переменную и использовать при обработке запроса:

IWebHostEnvironment \_env**;**public Startup**(**IWebHostEnvironment env**)  
{** \_env = env**;  
}**



# Middleware общее

Обработка запроса в ASP.NET Core устроена по принципу конвейера.

Сначала данные запроса получает первый компонент в конвейере. После обработки он передает данные HTTP-запроса второму компоненту и так далее. Эти компоненты конвейера, которые отвечают за обработку запроса, называются **middleware**. В ASP.NET Core для подключения компонентов middleware используется метод Configure из класса Startup.

Компоненты middleware конфигурируются с помощью методов расширений Run, Map и Use объекта **IApplicationBuilder**, который передается в метод Configure() класса Startup. Каждый компонент может быть определен как анонимный метод (встроенный inline компонент), либо может быть вынесен в отдельный класс.

Все вызовы типа app.UseXXX как раз и представляют собой добавление компонентов middleware для обработки запроса. То есть у нас получается примерно следующий конвейер обработки:

1. Компонент обработки ошибок - Diagnostics. Добавляется через app.UseDeveloperExceptionPage()
2. Компонент маршрутизации - EndpointRoutingMiddleware. Добавляется через app.UseRouting()
3. Компонент EndpointMiddleware, который отправляет ответ, если запрос пришел по маршруту "/" (то есть пользователь обратился к корню веб-приложения). Добавляется через метод app.UseEndpoints()

При этом порядок определения компонентов играет большую роль. Если мы изменим порядок, то приложение нормально работать не будет.

# Middleware components:

* Authentication: предоставляет поддержку аутентификации
* Cookie Policy: отслеживает согласие пользователя на хранение связанной с ним информации в куках
* CORS: обеспечивает поддержку кроссдоменных запросов
* Diagnostics: предоставляет страницы статусных кодов, функционал обработки исключений, страницу исключений разработчика
* Forwarded Headers: перенаправляет зголовки запроса
* Health Check: проверяет работоспособность приложения asp.net core
* HTTP Method Override: позволяет входящему POST-запросу переопределить метод
* HTTPS Redirection: перенаправляет все запросы HTTP на HTTPS
* HTTP Strict Transport Security (HSTS): для улучшения безопасности приложения добавляет специальный заголовок ответа
* MVC: обеспечивает функционал фреймворка MVC
* Request Localization: обеспечивает поддержку локализации
* Response Caching: позволяет кэшировать результаты запросов
* Response Compression: обеспечивает сжатие ответа клиенту
* URL Rewrite: предоставляет функциональность URL Rewriting
* Endpoint Routing: предоставляет механизм маршрутизации
* Session: предоставляет поддержку сессий
* Static Files: предоставляет поддержку обработки статических файлов
* WebSockets: добавляет поддержку протокола WebSockets

# Configure

## Run

Run представляет собой простейший способ для добавления компонентов middleware в конвейер. Однако компоненты, определенные через метод Run, не вызывают никакие другие компоненты и дальше обработку запроса не передают.

В качестве параметра метод Run принимает делегат RequestDelegate. Этот делегат имеет следующее определение:

public delegate Task RequestDelegate(HttpContext context);

так как данный метод не передает обработку запроса далее по конвейеру, то его следует помещать в самом конце.

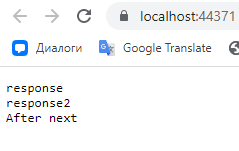
Если сделать 2 рана то будет вып. Только первый

## Use

Метод Use также добавляет компоненты middleware, которые также обрабатывают запрос, но в нем может быть вызван следующий в конвейере запроса компонент middleware. Например, изменим метод Configure() следующим образом:

app**.**Use**(**async **(**context**,** next**)** =>  
**{** await context**.**Response**.**WriteAsync**(**"response\n"**);** await next**.**Invoke**();** await context**.**Response**.**WriteAsync**(**"After next \n"**);  
});**app**.**Run**(**async **(**context**)** =>  
**{** await context**.**Response**.**WriteAsync**(**"response2\n"**);  
});**

Return:

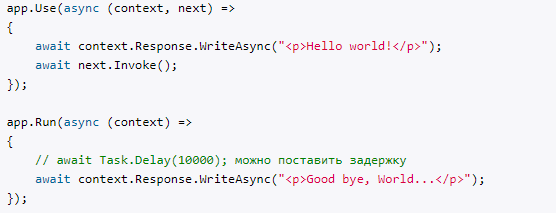


Однако в большинстве случаев мы можем использовать не просто методы Use, а методы расширений app.**UseXXX**, например, UseStaticFiles() или UseMvc().

**Но вообще не рекомендуется вызывать метод next.Invoke после метода Response.WriteAsync()**. Компонент middleware должен либо генерировать ответ с помощью Response.WriteAsync, либо вызывать следующий делегат посредством next.Invoke, но не выполнять оба этих действия одновременно

Так как согласно документации последующие изменения объекта Response могут привести к нарушению протокола, например, будет послано больше байт, чем указано в заголовке Content-Length, либо могут привести к нарушению тела ответа, например, футер страницы HTML запишется в CSS-файл.

Нинада так:



## Map()

Тоже MiddleWare, только для определенных path’ов

Метод Map (и методы расширения MapXXX()) применяется для сопоставления пути запроса с определенным 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 map

app**.**Map**(**"/home"**,** home =>  
**{** home**.**Map**(**"/index"**,** Index**);** home**.**Map**(**"/about"**,** About**);  
});**

## mapWhen()

Он принимает в качестве параметра делегат Func<HttpContext, bool>, если возращает true, то работает

public void Configure**(**IApplicationBuilder app**,** IWebHostEnvironment env**)  
{** app**.**MapWhen**(**context => **{** return context**.**Request**.**Query**.**ContainsKey**(**"id"**)** &&   
 context**.**Request**.**Query**[**"id"**]** == "5"**;  
 },** HandleId**);** app**.**Run**(**async **(**context**)** =>  
 **{** await context**.**Response**.**WriteAsync**(**"Good bye, World..."**);  
 });  
}**private static void HandleId**(**IApplicationBuilder app**)  
{** app**.**Run**(**async context =>  
 **{** await context**.**Response**.**WriteAsync**(**"id is equal to 5"**);  
 });  
}**

В данном случае если в запросе указан параметр id и он имеет значение 5, то запрос обрабатывается функцией HandleId(). К подобным запросам будут относиться, например, запрос *http://localhost:55234/?id=5* или *http://localhost:55234/product?id=5&name=phone*, так как обе строки запроса содержат параметр id равный 5. А все остальные запросы также будут обрабатываться делегатом, передаваемым в метод app.Run().

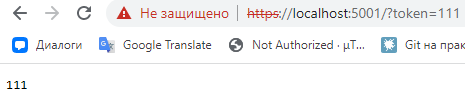
## UseMiddleware

Кроме использования делегатов в методах Run/Use/Map мы можем создавать свои компоненты middleware в виде отдельных классов, которые затем добавляются в конвейер с помощью метода UseMiddleware(). Например, напишем свой примитивный компонент. Для этого добавим в новый проект новый класс TokenMiddleware:

public class TokenMiddleware  
**{** private readonly RequestDelegate \_next**;** public TokenMiddleware**(**RequestDelegate next**)  
 {** this**.**\_next = next**;  
 }** public async Task InvokeAsync**(**HttpContext context**)  
 {** var token = context**.**Request**.**Query**[**"token"**];** if **(**token == "111"**)  
 {** context**.**Response**.**StatusCode = 403**;** await context**.**Response**.**WriteAsync**(**token**);  
 }** else  
 **{** await \_next**.**Invoke**(**context**);  
 }  
 }  
}**

//conf:

app**.**UseMiddleware<TokenMiddleware>**();**



Класс middleware должен иметь конструктор, который принимает параметр типа **RequestDelegate**. Через этот параметр можно получить ссылку на тот делегат запроса, который стоит следующим в конвейере обработки запроса.

Также в классе должен быть определен метод, который должен называться либо **Invoke**, либо **InvokeAsync**. Причем этот метод должен возвращать объект Task и принимать в качестве параметра контекст запроса - объект HttpContext. Данный метод собственно и будет обрабатывать запрос.

Часто пишут extentions:

public static class TokenExtensions  
**{** public static IApplicationBuilder UseToken**(**this IApplicationBuilder builder**,** string pattern**)  
 {** return builder**.**UseMiddleware<TokenMiddleware>**(**pattern**);  
 }  
}**

app**.**UseToken**(**"555555"**);**

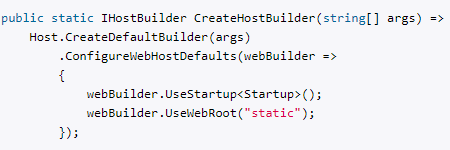
# static files

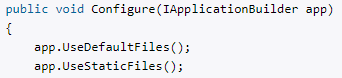
Мы можем хранить html странички изначально, они по умолчанию должны лежать в ContentRoot/WebRoot. По умолчанию "ContentRoot" представляет каталог текущего проекта, а "WebRoot" по умолчанию представляет каталог wwwroot.

Мы можем получить их так: [*http://localhost:58666/index.html*](http://localhost:58666/index.html)

Какая то фигня, согласен

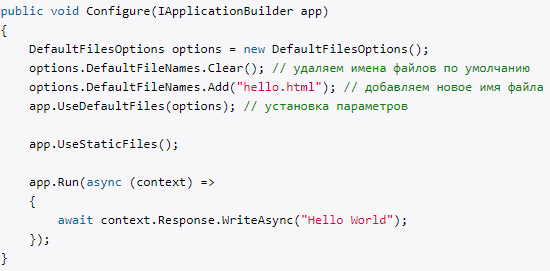
Но вообще на всяки,**папку можно изменять** с wwwroot на любую через useWebRoot:



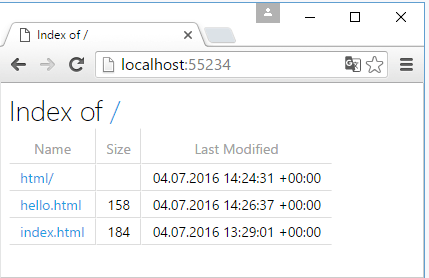
А еще с помощью UseDefaultFiles() **можно настроить отправку статических веб-страниц по умолчанию** без обращения к ним по полному пути: 

В этом случае при отправке запроса к корню веб-приложения типа *http://localhost:xxxx/* приложение будет искать в папке wwwroot следующие файлы:

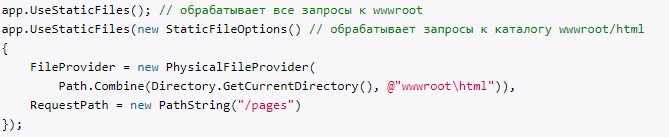
* default.htm
* default.html
* index.htm
* index.html

**Если же мы хотим использовать файл, название которого отличается от вышеперечисленных**, то нам надо в этом случае применить объект **DefaultFilesOptions**: 

Метод **UseDirectoryBrowser** позволяет пользователям просматривать содержимое каталогов на сайте:

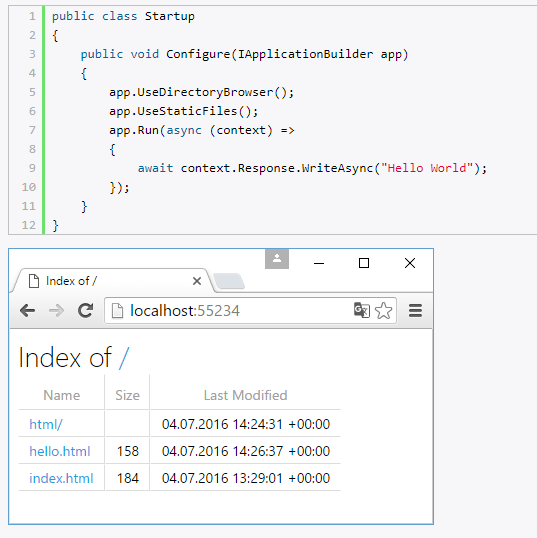


Перегрузка метода UseStaticFiles() позволяет **сопоставить пути с определенными каталогами:**



# UseDirectoryBrowser

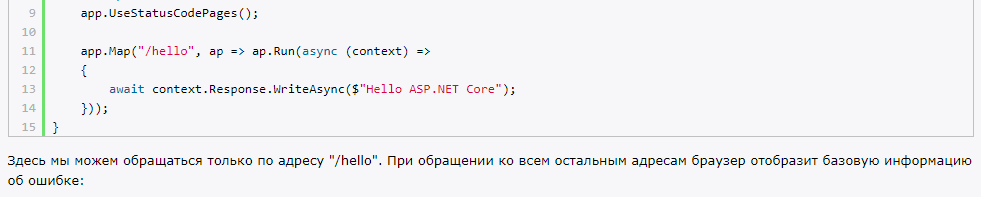
Метод **UseDirectoryBrowser** позволяет пользователям просматривать содержимое каталогов на сайте:

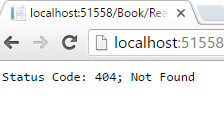


# App.UseDeveloperExceptionPage

Будет выдавать окно с инфой об ошибке

# app.UseStatusCodePages();

с помощью **StatusCodePagesMiddleware** можно добавить в проект отправку информации о статусном коде. Для этого добавим в метод Configure() класса Startup вызов app.UseStatusCodePages():



Данный метод позволяет настроить отправляемое пользователю сообщение. В частности, мы можем изменить вызов метода так:

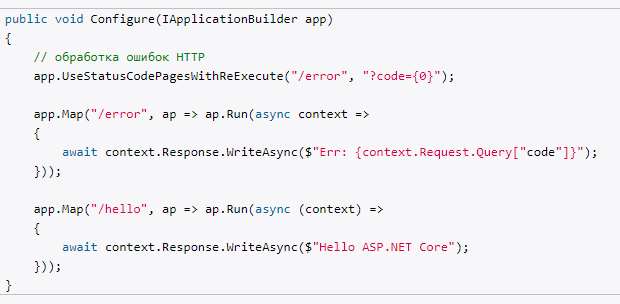
app.UseStatusCodePages("text/plain", "Error. Status code : {0}");

В качестве первого параметра указывается MIME-тип ответа, а в качестве второго - собственно то сообщение, которое увидит пользователь. В сообщение мы можем передать код ошибки через плейсхолдер "{0}".

**Можно еще redirectiть ошибки на определенный урл,** типо произошла ошибка, и мы отправляем код ошибки например на

app.UseStatusCodePagesWithRedirects("/error?code={0}");

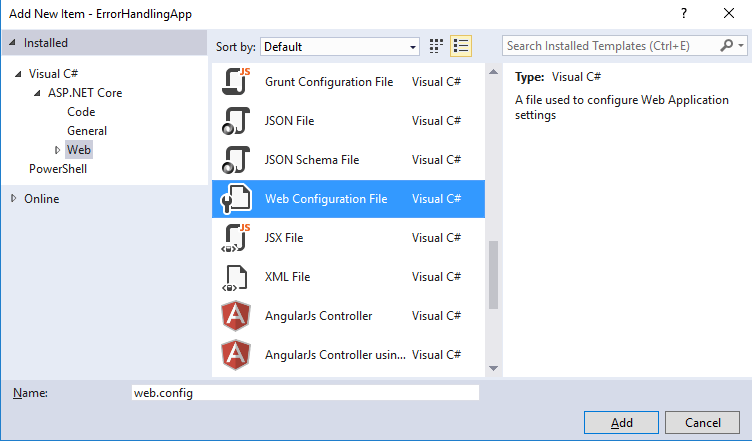
а по этому урлу мы уже можем обрабатывать всю эту суету



web.config

В ASP.NET Core он также доступен, однако имеет очень ограниченное действие. В частности, мы его можем использовать только при развертывании на IIS, а также не можем использовать ряд настроек.

Итак, добавим в корень проекта новый элемент **Web Configurarion File**, который естественно назовем **web.config**:



Итак, элемент httpErrors имеет ряд настроек. Для тестирования настроек локально, необходимо установить атрибут errorMode="Custom". Если тестирование необязательно, и приложение уже развернуто для использования, то можно установить значение errorMode="DetailedLocalOnly".

Значение existingResponse="Replace" позволит отобразить ошибку по оригинальному запрошенному пути без переадресации.

Внутри элемента httpErrors с помощью отдельных элементов **error** устанавливается обработка ошибок. Атрибут statusCode задает статусный код, атрибут path - адрес url, который будет вызываться, а атрибут responseMode указывает, как будет обрабатываться ответ вызванному url. Атрибут responseMode имеет значение File, что позволяет рассматривать адрес url из атрибута path как статическую страницу и использовать ее в качестве ответа

Настройки элемента httpErrors могут наследоваться с других уровней, например, от файла конфигурации machine.config. И чтобы удалить все унаследованные настройки, применяется элемент <clear />. Чтобы удалить настройки для отдельных ошибок, применяется элемент <remove />.