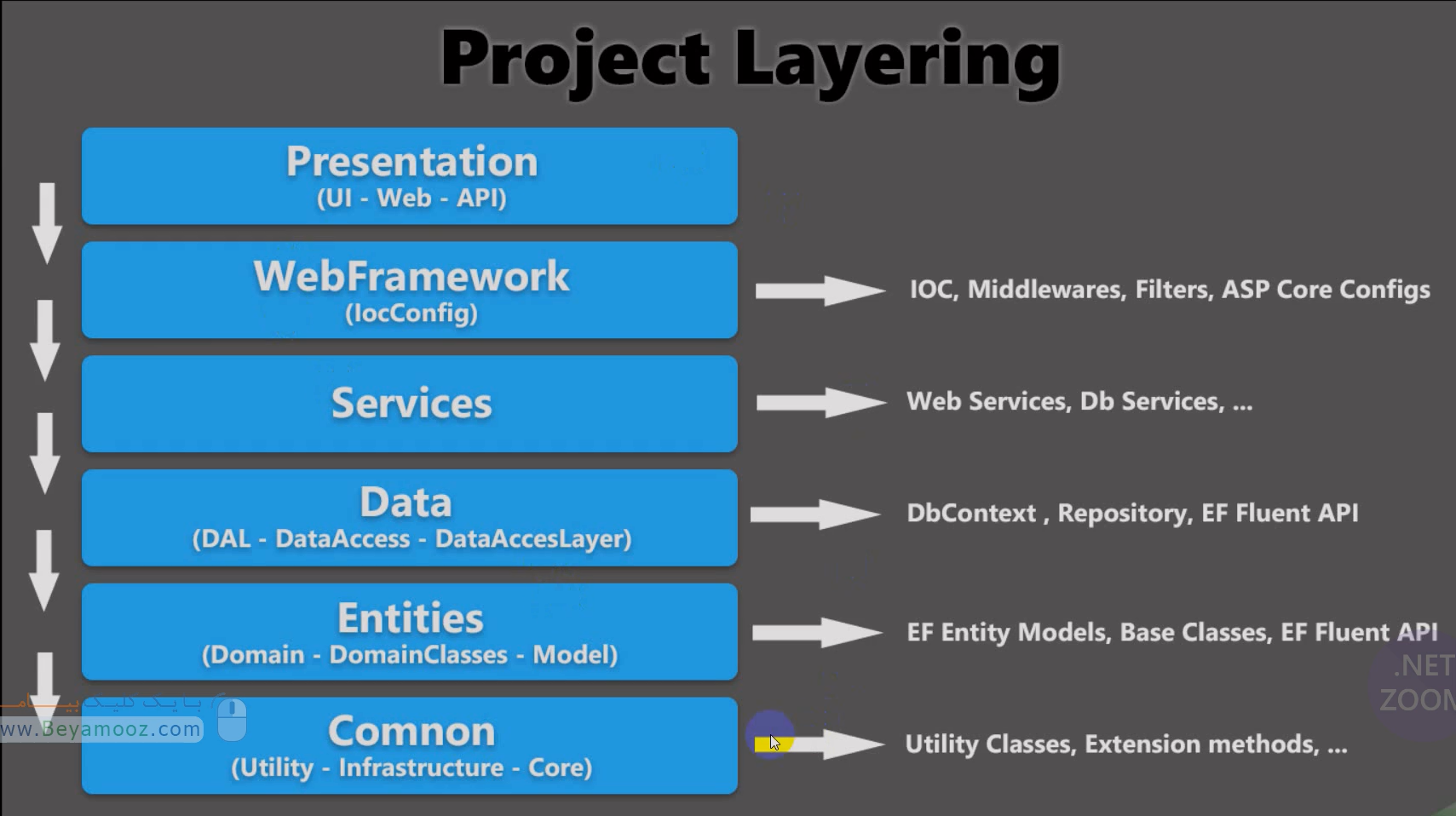
لایه بندی پروژه :



پایین ترین لایه، لایه Comnon هست، که با نام های Utility یا Infrastructure یا Core هم شناخته میشه. که در واقع ابزارهایی که در لایه های دیگه استفاده میشه در این لایه قرار میگیره. و معمولا وابستگی به پروژه نداره و میتونیم در پروژه های دیگه هم میتونیم ازش استفاده کنیم.

لایه بعدی Entities که با نام های Domain یا DomainClasses یا Model هم شناخته میشه. که مدل های پروژه داخلش قرار میگیره.

لایه بعدی Data هست که با نام های DAL یا ِDataAccess یا DataAccesLayer هم شناخته میشه. داخل این لایه هر چیزی که مربوط به لایه دسترسی به دیتا هست قرار میگیره.

لایه بعدی Services هست که سرویس های پروژه داخلش قرار میگیره. سرویس ها همون منطق تجاری پروژه ما هست که داخل سرویس پیاده سازی میشه. و این سرویس هست که میدونه از چه ریپازیتوری هایی باید استفاده کرد.

لایه بعدی WebFramwork هست که با نام IocConfig هم شناخته میشه. این لایه محل قرار گیری چیزهای مشترک بین لایه های web ما هست. در بعضی پروژه ها لایه web بیشتر از یکی هست مثلا یک لایه web برای api های پروژه یک لایه web برای ادمین پروژه و یک لایه web برای UI پروژه. حالا چیزهای مشترک این لایه ها رو داخل این لایه مینویسیم. مثل میدیل ویر ها یا فیلتر ها یا اتریبیوت هایی که لازم هست یا کانفیک های پروژه.

لایه Presentatio که با نام های UI یا Web یا API هم شناخته میشه. ولایه نهایی ما هست. تمای این لایه ها از لایه های زیرین خودشون استفاده میکنن.

نکته مهم : اگر لایه مثلا WebFramwork من داره از لایه Services داره استفاده میکنه و لایه Api من داره از WebFramwork من استفاده میکنه. در نتیجه Api من میتونه از Services استفاده کنه. این باعث میشه در هر لایه نیاز نباشه تمامی لایه های زیر رو رفرنس بدیم.

داخل لایه Common یک سری ابزار مینویسیم عبارت اند از :

کلاس Assert : برای اعتبارسنجی پارامترهای ورودی متد هست جایی که لازم هست پارامتری null نباشه واگر null باشه باید ArgumentNullException صادر بشه.

متد NotEmpty چک میکنه که ابجکتی خالی نباشه اگر از جنس String هست خالی نباشه و اگر لیست هست کانتش صفر نباشه.

کلاس EnumExtension که برای کار با Enum ها هست. اینکه یک Enum میگیریم و مقادیر داخلش رو به صورت IEumerable دریافت میکنیم. یا اینکه یک Enum میگیریم و مقادیر داخلش رو با Flags چک میکنیم.

متد ToDisplay که داخلش از متد DisplayAttribute استفاده میکنیم که این متد یه سری اتریبیوت داره که ازش استفاده میکنیم. و زمانی که بالای پروپرتی ها از اتریبیوت display استفاده میکنیم کافیه که نام پارامتر رو بدیم به این متد و مقداری که در display ش قرار گرفته رو بگیریم ازش.

کلاس IdentityExtension هست. زمانی که از Identity استفاده میکنیم به کار میاد. میتونیم مقادیر داخل ClaimsIdentity رو خارج کنیم. توسط اسم کلیم.

کلاس ModelBuilderExtension که برای تنظیمات entity هامون به صورت Reflection هست.

لایه Entities :

آیدی در همه مدل ها استفاده میشه و برای اینکه نخوایم در همه کلاس ها این پروپرتی رو بنویسیم از یک کلاس به نام BaseEntity استفاده میکنیم و این فیلد رو داخلش مینویسیم :

Public abstract Class BaseEntity<Tkey> {

Public TKey Id {get; set; } }

به دلیل اینکه کلاس ها ایدیشون از نوع های مختلفی هست کلاس baseEntity رو به این صورت نوشتیم و چون اکثر ایدی ها از جنس int هست یک کلاس دیگر به صورت زیر درست میکنیم:

Public abstract Class BaseEntity : BaseEntity <int> {}

برای اینکه بخوایم از بحث رفلکشن استفاده کنیم و اینکه کلاس های خودمون رو بشناسیم و مارکشون کنیم به نحوی که بعدا قابل تشخیص باشند میایم از یک اینترفیس استفاده میکنیم. و به این دلیل از baseEntity استفاده نمیکنیم که بعضی از کلاس ها به ایدی نیاز ندارند و از baseEntity هم ارث بری نمیکنند.

Public interface IEntity {}

حالا کلاس BaseEntity جنریک رو از IEntity ارث بری میکنیم.

استفاده از FluentApi : برای جلوگیری از استفاده از اتریبیوت بالای سر پروپرتی های انتیتی استفاده میشه.

مثلا برای انتیتی post :

Public Class PostEntityMap : IentityTypeConfiguration <post> {

Public void Configure (EntityTypeBuilder <Post> builder ) {

builder.Property (\_ => \_.Title).IsRequired().HasMaxLength (200);

builder.Property (\_ => \_.Description).IsRequired ();

builder.HasOne (\_ => \_.Category)

.WithMany (\_ => \_.Posts).HasForeignKey (\_ => \_.CategoryId);

builder.HasOne (\_ => \_.Author)

.WithMany (\_ => \_.Posts).HasForeignKey (\_ => \_.AuthorId);

} }

ساخت لایه DbContext :

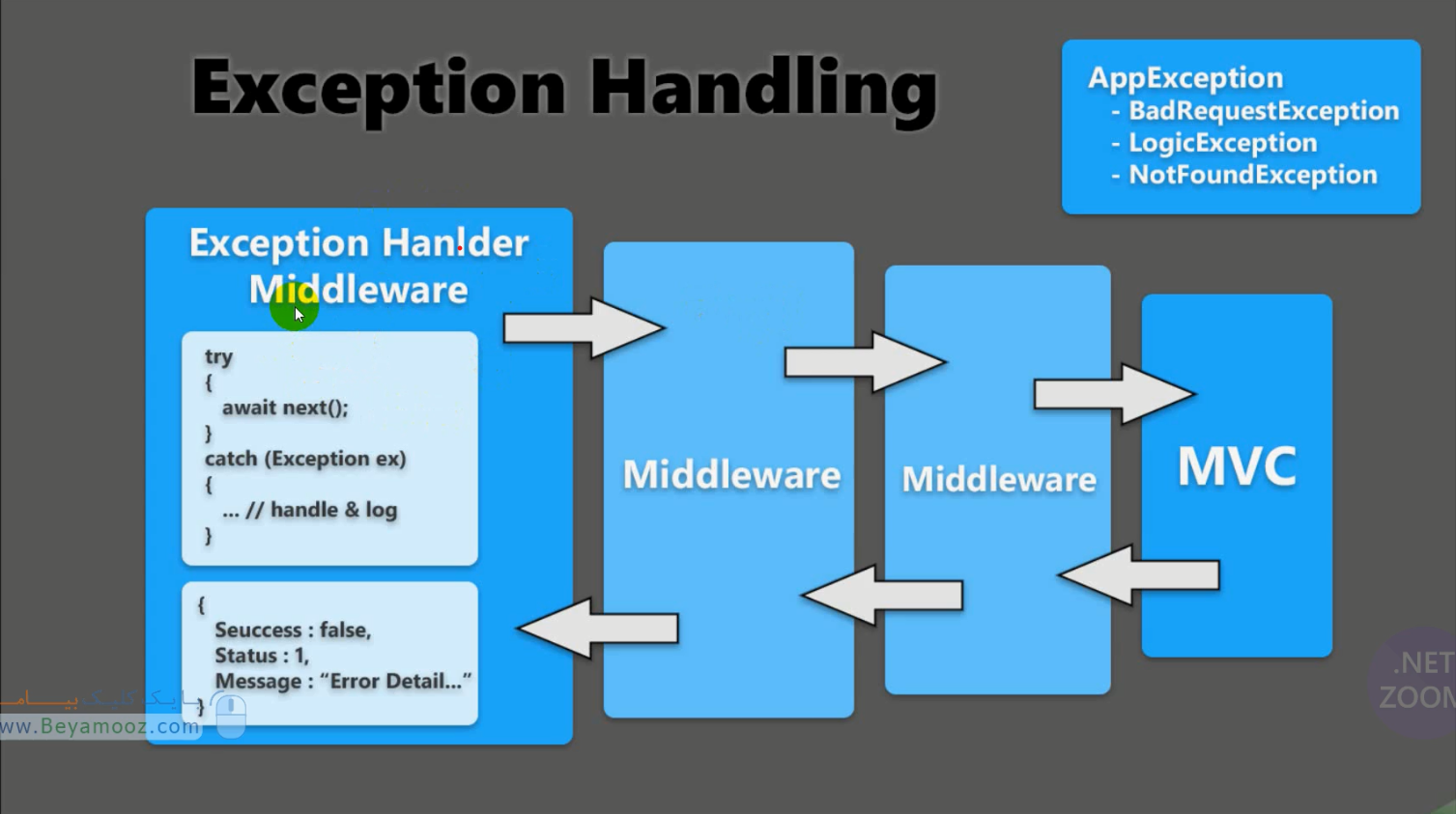
استفاده از رفلکشن به جای تعریف کلاس ها در این کلاس. میخایم به صورت اتوماتیک کلاس هایی که از IEntity

متد OnModelCreating وظیفه ساخت جدول از روی مدل های ما رو داره.

Public Class ApplicationDbContext : DbContext {

آموزش Exception Handling

Exception Handling یعنی اینکه خطاهایی که در سیستم رخ میدهد و مدیریت نشده است رو به دام بندازیم و کاری که میخایم رو روشون انجام بدیم. مثلا یه جا لاگش کنیم یا در دیتابیس ذخیرش کنیم و ...



در تصویر بالا لیست Request Pipeline ما وجود داره: به معنی خط لوله درخواست ها لیستی از میدیل ویر ها هستند که درخواست ما اول میاد داخل اولی بعد دومی بعد سومی ... و به یه جایی میرسه که از اونجا برگشت پیدا میکنه و این مسیر رو برمیگرده برعکس ادامه میده یعنی از سومی به دومی و بعد به اولی.

میدیل ویر ها به این صورت هست که داخل خودشون میدیل ویر ها رو صدا میزنن یعنی میدیل ویر اول داخل خودش میدیل ویر دوم رو صدا میزنه میدل ویر دوم داخل خودش میدیل ویر سوم رو صدا میزنه و به همین ترتیب.

Mvc هم یک میدیل ویر برای asp.net core هست. که میدیل ویر بزرگی هم هست.

ما برای exception handling یه میدیل ویر درست میکنیم که داخلش این کار رو انجام میدیم. و اون رو قبل از همه میزاریم یعنی در تصویر اولین میدل ویر مربوط به exception handling هست. به این خاطر اول قرارش دادیم که در هرجایی از برنامه خطایی به وجود اومد بتونیم گیرش بندازیم.

اگر در محیط توسعه بودیم نمایش جزئیات ارور مشکلی نداره ولی اگر در محیط سرور بودیم باید یه پیغام مناسب نمایش بدیم و نباید جزئیات رو نمایش بدیم.

ما معمولا سه نوع خطا داریم که بیشتر رخ میدهند:

1. Bad Request Exception : اینکه مقادیر وقتی از کلاینت به سمت سرور ارسال میشه نخایم مدام مدل استیت یا Is valid رو چک کنیم این کار رو ما انجام نمیدادیم توسط Attribute مربوط به کنترلر.
2. Not Found Exception : وقتی که کاربر چیزی رو میخاد که وجود نداره.
3. Logic Exception : بحث خطا توی منطق تجاری پروژه ما هست. مثلا یه دیتایی که سمت ما میاد و ما میخایم یه وب سرویس رو صدا بزنیم و اون وب سرویس به هر دلیلی خطا میدهد.

همه این اکسپشن ها هم زیر مجموعه app exception هستند و اگر خطایی بود که از هیچ کدام از این سه مدل نبود میتونیم از app exception استفاده کنیم.

اول از همه میدیل ویر خودمون رو میسازیم:

توضیح متد های Startup :

متد Configure Services : این متد سرویس هایی رو به IService Collection اضافه میکنه.

IService Collection هم یک لیست از Service Descriptor هست.

Service Descriptor هم یعنی توضیح دهنده یک سرویس.

متد Configure : وظیفش این هست که میدیل ویر ها رو به پروژه ما اضافه کنه و در واقع Request Pipeline ما رو میسازه.

به سه روش میتونیم میدیل ویر اضافه کنیم:

App.Use

App.Run

App.UseMiddleware

میدیل ویر خودمون رو در یک کلاس به صورت اکستنش مینویسیم و بعد در متد Configure صداش میزنیم.

نکته : خطاهای مدیریت نشده یعنی خطاهایی که در برنامه پیش بینی براش در نظر گرفته نشده.

اسم کلاس خودمون رو میزاریم : Custom Exception Handler Middleware

Public Class CustomExceptionHandlerMiddleware

{

Private readonly RequestDelegate next;

Private readonly ILogger<CustomExceptionHandlerMiddleware> logger;

Public CustomExceptionHandlerMiddleware (

RequestDelegate next,

ILogger<CustomExceptionHandlerMiddleware> logger)

{

این نکست در واقع اشاره میکنه به میدیل ویر بعدی This.next = next;

}

Public async Task Invoke (HttpContext httpContext)

{

Try

{

Await next(httpContext);

}

Catch(Exception exception)

{

Logger.LogError(exception, “khataii rokh dade”);

Var apiResult = new ApiResult (false,

ApiResultStatusCode.ServerError);

Var json = JsonConvert.SerializeObject(apiResult);

Await httpContext.Response.WriteAsync (json);

}

}

}

با کدهای بالا خطاها رو لاگ میکنیم. اول میام خطا رو به دام میندازیم بعد در قالب یک api result درش میاریم و فرستادیمش به سمت کلاینت.

حالا میخایم نوع خطایی که رخ داده هم نمایش بدیم ( جزئیات خطا) برای همین Custom Exception ها رو اضافه میکنیم.

برای این کار باید چند تا Custom exception مینویسیم. یک AppException اضافه کنیم اول که بقیه Exception ها از اون ارث بری میکنن.

Public Class AppException : Exception

{

Public ApiResultStatusCode statusCode { get; set; }

Public AppException (String message, ApiResultStatusCode statusCode)

: base (message)

{

statusCode = statusCode;

}

}

هدف ما این هست که یک Custom Exception درست کنیم که بتونیم پیغام مورد نظر رو بهش پاس بدیم همچین استوس کد مورد نظرمون رو بهش پاس بدیم.

حالا BadRequestException :

Public Class BadRequestException : AppException

{

Public BadRequestException (string message, ApiResultStatusCode statusCode)

: base (message, ApiResultStatusCode.BadRequest)

{

}

}

حالا NotFoundException :

Public Class NotFoundException : AppException

{

Public NotFoundException(string message)

: base (message, ApiResultStatusCode.NotFound)

{

}

}

حالا LogicException :

Public Class LogicException : AppException

{

Public LogicException (string message)

: base (message, ApiResultStatusCode.LogicError)

{

}

}

حالا میریم داخل میدیل ویر کاستومی که ساختیم متد Invoke و :

Catch (AppException ex)

{

Logger.LogError (ex, ex. message)

Var apiResult = new ApiResult (false, ex.StatusCode);

Var json = JsonConvert.SeriallizeObject (apiResult);

Await httpContext.Response.WriteAsync (json);

}

ترتیب نوشتن این اکسپشن ها مهم هست. مثلا اگر اکسپشنی که شامل همه نوع خطایی میشود را اگر اول بنویسیم همیشه میره داخل این اکسپشن و اکسپشن های بعدی را چک نمیکنه.

نکته : این میدل رو باید به لیست میدیل ویرهامون اضافه کنیم.

App.UseMiddleware < CustomExceptionHandlerMiddleware > ();

یا اینکه به اکستنشن تبدیلیش کنیم :

Public Static Class CustomExceptionHandlerMiddlewareExtensions

{

Public static void UseCustomExceptionHandler (this IApplicationBuilder app)

{

App.UseMiddleware < CustomExceptionHandlerMiddleware > ();

}

}

حالا به جای دستور بالا :

App.UseCustomExceptionHandler();

پروژه که اجرا میکنیم اول میره تو try و میدیل ویر های بعدی رو صدا میزنه اگر که در اون میدیل ویر ها خطایی رخ بده میاد سروقت catch هایی که تنظیم کردیم.

حرفه ای تر کردن Exception handling :

در کلاس AppException :

Public Class AppException : Exception

{

Public HttpStatusCode HttpStatusCode { get; set; }

Public ApiResultStatusCode ApiStatusCode { get; set; }

Public object AddittionalData { get; set; }

Public AddException() : this ( ApiResultStatusCode.ServerError )

{

}

}

HttpStatusCode برای این هست که میخایم یک StatusCode رو برگردونیم ولی یک HttpStatusCode خاص رو میخایم در کنارش برگردونیم.

AdditionalData یا اینکه در کنار اون استاتوس یک سری دیتای اضافه پاس بدیم به اکسپشنمون و موقع نمایش اون دیتاها رو هم نمایش بدیم.

و یک سری کانستراکتور هم به این کلاس اضافه کردیم:

Public AppException ( ApiResultStatusCode statusCode, string message,

HttpStatusCode httpStatusCode, Exception exception, object additionalData)

: base (message, exception)

{

ApiStatusCode = statusCode;

HttpStatusCode = httpStatusCode;

AdditionalData = additionalData;

}

این کانستراکتور کامل ترین کانستراکتور هست که در ورودی استاتوس کد رو میگیره، پیغام خطا، httpStatusCod، اکسپشنی که مدیریت شده هم هست رو بهش میدیم و در قالب یک AppException میده به کلاینت، و دیتاهای اضافی هم در نهایت ازمون میگیره.

توجه : این سازنده اومده بیس رو صدا زده در صورتی که بقیه سازنده ها خود کلاس AppException رو صدا میزنن. مثال :

Public AppException ( ApiResultStatusCode statusCode, string message, HttpStatusCode httpStatusCode) : this (statusCode, message, httpStatusCode, null)

{

}

Public AppException ( ApiResultStatusCode statusCode) : this (statusCode, null)

{}

Public AppException ( string message )

: this ( ApiResultStatusCode.ServerError, message) {}

مشابه این روش برای BadRequestException هم پیاده سازی میکنیم و همه exception ها از سازنده پدر ارث بری میکنند به جز سازنده اخر :

Public BadRequestException (string message, Exception exception,

Object additionalData) : base (ApiResultStatusCode.BadRequest, message,

Exception, additionalData)

{}

برای NotFoundException و LogicException هم متداورلودیگ های مختلف مینویسیم.

این باعث میشود کلاس ها انعطاف پذر باشند و در هر موقعیت بتوانیم از متد به روش دلخواه استفاده کنیم.

حالا کلاس CustomExceptionHandlerMiddleware هم کامل تر میکنیم:

Public Class CustomExceptionHandlerMiddleware

{

Private readonly RequestDelegate \_next;

Private readonly IHostingEnvironment \_env;

Private readonly ILogger <CustomExceptionHandlerMiddleware> \_logger;

Public CustomExceptionHandlerMiddleware (RequestDelegate next,

IHostingEnvironment emv,

ILogger <CustomExceptionHandlerMiddleware> logger)

{

\_next = next;

\_env = env;

\_logger = logger;

}

IHostingEnvironment : که environment ، development یا Production رو میتونیم از روش به دست بیاریم.

حالا در متد Invoke :

Public async Task Invoke (HttpContext context)

{

String message = null;

HttpStatusCode httpStatusCode = HttpStatusCode.InternalServerError;

ApiResultStatusCode apiStatusCode = ApiResultStatusCode.ServerError;

Try

{ await \_next (context) ; }

Catch (AppException exception)

{

\_logger.LogError (exception, exception.Message);

httpStatusCode = exception.HttpStatusCode;

apiStatusCode = exception.ApiStatusCode;

if (\_env.IsDevelopment())

{

Var dic = new Dictionary <string, string>

{

[“Exception”] = exception.Message,

[“StackTrace”] = exception.StackTrace

}

If (exception.InnerException != null)

{

dic.Add (“InnerException.Exception”, exception.InnerException.Message);

dic.Add

(“InnerException.StackTrace” , exception.InnerException.StackTrace);

}

If (exception.AdditionalData != null)

{

dic.Add (“AdditionalData”, JsonConvert.SerializeObject (exception.Additionaldata));

}

اگر که در محیط توسعه بودیم با کدهای بالا میام جزئیات خطا و اینکه خطا در چه خطی رخ داده.

یا اینکه اگر InnerException خالی نیست بیا جزئیات innerException هم بهمون نشون بده.

برای اینکه کار اول یه دیکشنری درست کردیم. و با اطلاعاتی که از خطا به دست میاریم میخایم پرشون میکنیم.

و اینکه اگر دیتایی داریم میفرستیم با اکسپشن بیا و اونها رو هم اضافه کن.

و نهایتا :

Message = JsonConvert.SerializeObject (dic);

همه دیتا رو به جیسان تبدیل کن و بفرست به کلاینت.

ولی اگر در محیط توسعه نبودیم :

Else {

Message = exception.Message;

}

و نهایت یک متد نوشتیم برای write کردن به نام WriteToResponseAsync.

یک اکسپشن برای jwt : زمانی که توکن ما منقضی شده

Catch (SecurityTokenExpiredException exception)

{

\_logger.LogError (exception, exception.Message);

setUnAuthorizeResponse (exception);

await WriteToResponseAsync();

}

اول یک لاگ ازش گرفتیم بعد یه سری اطلاعات ازش استخراج کردیم و نهایتا اون رو توی response رایت کردیم.

یک اکسپشن برای jwt زمانی که میخایم به یک متدی دسترسی داشته باشیم که آتورایز داره ولی توکنی ارسال نکردیم :

Catch (UnauthorizedAccessException exception)

{

\_logger.LogError (exception, exception.Message);

SetUnAuthorizeResponse (exception);

await WriteToResponseAsync();

}

حالا متد WriteToResponseAsync :

Async Task WriteToResponseAsync()

{

If (context.Response.HasStarted)

{

Throw new InvalidOperationException (

“ The response has already started, the status code “);

Var result = new ApiResult (false, apiStatusCode, message);

Var json = JsonConvert.SeralizeObject (result);

Context.Response.StatusCode = (int) httpStatusCode;

Context.Response.ContentType = “application/json”;

await context.Response.WriteAsync (json);

}

این متد وظیفش این هست که خروجی رو چاپ کنه. اول میاد چک میکنه که اگر ریسپانس ما قبلا استارت شده باشه ارور برمیگردونه. دلیلش هم این هست که قاعدتا زمانیکه اکسپشن رخ میده ریسپانش قبلا رایت نشده باشه واگر ریسپانس قبلا رخ داده باشه یه اتفاق غیرمعمول است و باید اکسپشن برگردونیم.

و بعد با توجه به اطلاعاتی که گرفتیم میایم ApiResult مورد نظرمون رو درست میکنیم. و بعد به json تبدیلش میکنیم. و بعد هم توی خروجی چاپش میکنیم.

و متد SetUnAuthorizeResponse :

کار این متد این هست که یه سری جزئیات بیشتر رو از بدنه exception بیرون میکشه و میریزه داخلش چنتا پروپرتی که ست کردیم مثل :

httpStatusCode , apiStatusCode, message

void SetUnAuthorizeResponse (Exception exception)

{

httpStatusCode = HttpStatusCode.Unauthorized;

apiStatusCode = ApiResultStatusCode.UnAthorized;

if (\_env.IsDevelopment())

{

Var dic = new Dictionary <string, string>

{

[“Exception”] = exception.Messge,

[“StackTrace”] = exception.StackTrace

};

If (exception is SecurityTokenExpiredException tokenException)

dic.Add (“Expires”, tokenException.Expires.ToString());

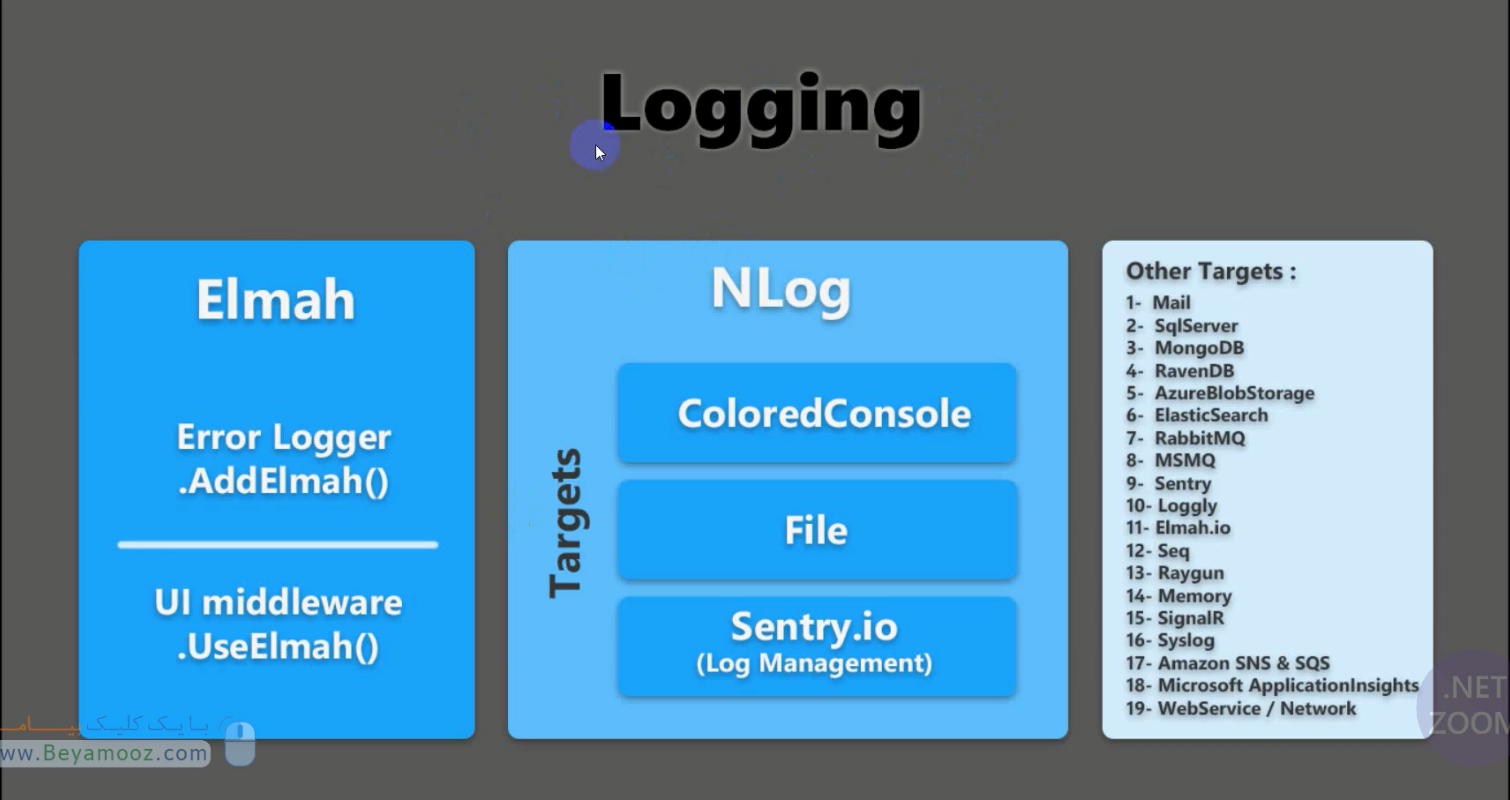
message = JsonConvert.SerializeObject (dic);

}

}

درون if میگه که اگر داخل محیط توسعه هستیم بیا جزئیات بیشتری بهمون نشون بده و اگر توکن ما Expire شده بیا و زمان انقضا رو بهمون نشون بده. و نهایتا دیتا رو به صورت جیسان داخل message ریختیم.

Exception Logging :



لاگینگ یعنی لاگ کردن که منحصرا به لاگ کردن اکسپشن هم ختم نمیشه و میتونیم اطلاعات یا یه چیزای دیگه هم لاگ کرد.

Elmah : یک ابزار لاگ گیری است که علاوه بر لاگ کردن یه ابزار در اختیارمون قرار میده که لیست خطاها رو نشون میده و جزئیات خطا هم بهمون نشون میده. که از دو بخش تشکیل شده : بخش اول سرویس ها و لاگر لازم برای لاگ کردن خطا هست و بخش دوم یک میدیل ویر هست که باعث میشه یک ui به ما بده که همون داشبوردی هست که خطاها رو بهمون نمایش میده. ولی برای پروژه های بزرگ مناسب نیست و بدرد پروژه های کوچیک و متوسط میخوره. چرا که داشبورد ساده در اختیار ما قرار میزاره و امکانات گزارش گیری از خطاها یا نمودار نداره.

Nlog :

یه مفهومی داره به نام Target که مقصدهایی هست که پروژه لاگ ها رو اونجا ذخیره میکنه.

یک Colored Console که یه کونسولی هست که شبیه صفحه مشکی رنگ هست با این تفاوت که متن ها رنگی نمایش میده.

دو File که داخل فایل ذخیره میکنه

سه Sentry.io که برای اینکه بخایم لاگ ها رو داخل دیتابیس ذخیره کنیم از این ابزار استفاده میکنیم.

که یک log management آنلاین هست که بجای اینکه داخل دیتا بیس اصلی یا یک دیتابیس جدا استفاده کنیم. لاگ ها رو داخل این ابزار ذخیره میکنیم.

توی nlog تارگت های خیلی زیادی هست که در بخش سوم تصویر تارگت های مختلف رو میتونیم ببینیم.

استفاده از Elmah در برنامه :

اول کتابخونه های لازم رو باید اضافه کنیم:

ElmahCore

ElmahCore.Sql

در ساده تراین حالت استفاده از Elmah :

در متد ConfigureServices :

Services.AddElmah();

و در متد Configure :

app.UseElmah();

و لاگ ها رو داخل مموری ذخیره میکنه و به محضه اینکه برنامه رو استپ میکنیم لاگ های ایجاد شده از بین میره.

ادرس پیش فرض Elmah : بعد از اینکه url پروژه رو وارد کردیم /elmah رو اضافه کنیم.

ذخیره لاگ ها در دیتا بیس :

Services.AddElmah (options =>

{

Options.Path = “/elmah-errors”;

}

با دستور بالا آدرس صفحه داشبورد خطاها به url / elmah-errors تغییر میکنه.

حالا باید یک جدول ایجاد کنیم که قرار هست خطاها داخلش ثبت بشه. و بعد هم بیایم کانکشن استریک رو تعریف کینم.

وارد گیت هاب مربوط به Elmah میشیم و فایل Sql ای مربوط به Elmah رو دانلود میکنیم. و کوئری که دانلود کردیم رو روی Sql، run میکنیم.

ستون های این جدول عبارت اند از :

ErrorId Application Host Type Source Message User StatusCode TimeUtc Sequence AllXml

حالا باید کانکشن اضافه کنیم:

Services.AddElmah <SqlErrorLog> (options =>

{

Options.path = “/elmah-errors”;

Options.ConnectionString = Configuration.GetConnectionString (“Elmah”);

در appsettings :

“ConnectionStrings” : {

“Elmah” : “Data Source=.;Initial Catalog = myApiDb; Integrated Security = true;”

}

اگر پروژه لاگ بده در دیتابیس ذخیره میشه.

متد RiseError : با این متد میتونیم یه دیتایی رو لاگ کنیم:

Public async Task<user> Create(userDto dto)

{

HttpContext.RiseError (new Exception (“method create farakhani shod”));

..

با این متد میتونیم اکسپشن هامون رو لاگ کنیم.

حالا اگر متد پست رو صدا بزنیم این متن اکسپشن به صورت لاگ شده در دیتا بیس ذخیره شده.

مشکلات Elmah :

یک : سازگاری کاملی با .net core نداره و با متد Ilogger نمیتونیم لاگ Elmah رو ثبت کنیم.

دو : پرفرمنس پایینی داره نسبت به لاگر های دیگه.

سه : داشبورد ساده و کم امکاناتی داره و بدون قابلیت جستجو هست.

NLOG :

اول باید پکیج لازم رو نصب کنیم:

Nlog.Web.AspNetCore

Nlog.Targets.Sentry2

پکچ دوم برای استفاده از Sentry برای نمایش لاگ ها هست.

برای استفاده از nlog باید داخل متد Program کد زیر رو اضافه کنیم:

Public Static IWebHostBuilder CreateWebHostBuilder (string[] args) =>

WebHost.CreateDefaultBuilder (args)

.ConfigureLogging (options => options.clearProviders())

.UseNlog()

.UseStartUp <Startup>();

چون به صورت پیش فرض دات نت کور از تارگت های مختلفی برای نمایش لاگ ها استفاده میکنه ( مثل کنسول) برای اینکه این تارگت های پیش فرض رو پاک کنیم و تارگت خاص خودمون رو اضافه کنیم از دستور :

clearProviders()

استفاده کردیم.

حالا کار دیگه ای که باید انجام بدیم این هست که یک فایل با نام Nlog.Config رو به پروژه اضافه کنیم. این فایل شامل تمام تنظیماتی هست که برای nlog استفاده میشه.

این فایل از جنس Xml هست. یک internalLogFile داره که ادرس فایل رو از ما میگیره.

internalLogFile = “d:\temp\internal-nlog.log”

فرمتی که توی فایل ذخیره میکنه به صورت متن هست.

این فایل مسئول خطاهای داخلی هست که داخل nlog ممکنه پیش بیاد. بلخره خود nlog یک کتابخونه هست که ممکنه داخل خودش خطاهایی رخ بده مثل اینکه کانفیک مربوط به nlog رو اشتباه تایپی داخلش داشته باشیم.

قسمت بعدی :

<extensions>

<add assembly = “Nlog.Web.AspNetCore”/>

<add assembly = “NLog.Targets.Sentry” />

</exceptions>

اینجا باید لیست ماژول هایی که قرار به nlog اضافه بشن رو بنویسیم.

<targets async = “true”>

<target xsi:type=”File” name=”LogFile” filename=”d:\temp\nlog-${shortdate}.log” layout=”${longdate} | ${uppercase:${level}} | ${message} | ${exception:format=tostring} | ${ogger} | url: ${aspnet-request-url} | action: ${aspnet-mvc-action}” />

<target xsi:type=”ColoredConsole” name=”ColorConsole” layout=”${longdate} | ${level} | ${exception:format-type} :

<target xsi:type=”sentry” name=”Sentry” dsn=”https://a4858522c:s58258

</targets>

اینجا مقصدهایی هست که میخایم لاگ ها رو داخلشون ذخیره کنیم.

اولی تایپش فایل هست که میتونه تایپ های مختلفی میتونیم بنویسیم. نام فایل هم بهش میدیم و بعد مسیر فایل رو بهش میدیم و بر اساس زمان فایل های مختلفی رو برامون میسازه. و همه لاگ ها رو داخل یک فایل ذخیره نمیکنه.

Layout قالبی هست که خطاها توی اون قالب باید فرمت بشن و نوشته بشن. اول گفتم که longdate تاریخ روز ماه سال رو بنویس بعد یه پایپ لاین بزار بعد بیا لول ارور رو بنویس بعد یک پایپ لاین بعد متن خطا بعد پایپ لاین بعد میاد اکسپشن رو به صورت استرینگ ذخیره میکنه بعد پایپ لاین بعد اسم لاگر رو مینویسه بعد پایپ لاین بعد اسم url ای که زمانی این خطا اتفاق افتاده چی بوده بعد پایپ لاین و در نهایت اکشنی که درخواست شده بوده و این خطا رخ داده بوده چی بوده. تعداد این ها زیاده و میتونیم چیزای دیگه ای هم اضافه کنیم.

تارگت دوم به صورت رنگی لاگ ها رو نشون میده به صورت زرد و خاکستری و سفید و قرمز و... و کنسول هست یه اسم بهش میدیم و بعد layout مون رو باید مشخص کنیم.

و تارگت سوم تایپ Sentry هست که توسط ماژول Sentry اضافه شده نامش رو مشخص کردیم. تارگت sentry یک آدرس dsn ازمون میگیره.

Dsn یک ادرس https هست که مثل توکن عمل میکنه و باعث میشه خطا ها و لاگ ها در این مقصد برامون ذخیره کنه و ما لازم نیست که ادرس رو باز کنیم تا این خطا ها رو ببینیم بلکه پروفایل خودمون رو داخل سایت sentry باز میکنیم. و این dsn رو از خود سایت sentry دریافت میکنیم.

<rules>

<logger name=”\*” minlevel=”Info” writeTo=”ColorConsole” />

<logger name=”\*” minlevel=”warn” writeTo=”LogFile” />

<logger name=”\*” minlevel=”Error” writeTo=”Sentry” />

</rules>

Rule ها یک سری شرایط رو برای تارگت هایی که نوشتیم مشخص میکنه. مثلا قسمت writeTo میاد اسم تارگت رو ازمون میگیره. Minlevel برمیگره به level خطا و میاد حداقل لولی که از اون لول به بالا لاگ ها میخایم ذخیره بشند رو مشخص میکنیم.

مثلا توسط دستور اول گفتیم حداقل لولی که میخایم ذخیره بشه info هست یعنی پایینتر از لول info که میشه debug ذخیره نمیشه.

قسمت async =”true” در تارگت باعث میشه که عملات ثبت لاگ به صورت async انجام بشه و در نتیجه اگر ثبت لاگ ما مثلا 10 ثانیه طول بکشه انجام عملات متوقف نمیشه و پروژه به کار خودش ادامه میده.

و بیشتر وقتی به کار میاد که داریم از تارگت شبکه ای مثل sentry استفاده میکنیم.

همچنین میتونیم یه دیفالت لیوت درست کنیم و در قسمت layout اسم اون رو قرار بدیم.

نکته : باید روی فایل nlog.Config کلیک راست کنیم و properties و بعد در قسمت

Copy To OutPut Directory : Copy Always

قرار میدیم. اگر این کار رو نکنیم nlog اصلا کار نمیکنه. این کار باعث میشه با هر bulid فایل nlog ما کپی بشه داخل پوشه bin پروژه و حالا nlog میتونه تنظیماتش رو بخونه ولی اگر این کار رو انجام ندیم nlog ما کپی نمیشه و کار نمیکنه.

و حتما هر تغییری دادیم یک ریبید از پروژه بگیریم که مطمعن بشیم که تغییرات ذخیره شده.

و حالا کار بعدی :

Elmah بعد از run شدن هاست و وقتی که همه چیز اوکی بود از اون لحظه به بعد میتونه خطاها رو لاگ کنه ولی اگر خطایی توی متد main کلاس Program رخ بده Elmah به هیچ عنوان نمیتونه اون رو لاگ کنه ولی nlog این کار رو هم انجام میده.

حالا به جای اینکه کد درون main رو به این صورت بنویسیم :

CreateWebHostBuilder (args).Build().Run();

به این صورت بنویسیم :

WebRequest.DefaultWebProxy = new WebProxy (“http://127.0.0.1.8118”, true) {

useDefaultCredentials = true

var logger = NLogBuilder.ConfigureNLog (“nlog.config”).GetCurrentClassLogger();

}

Try {

Logger.Debug (“init main”);

CreateWebHostBuilder (args).Build().Run(); }

Catch (Exception ex) {

Logger.Error (ex, “Stopped program because of exception”);

Throw; }

Finally {

NLog.LogManager.Shutdown (); }

خط اول برای این هست که بتونیم از Sentry استفاده کنیم چون سایت تحریم هست اومدیم Proxy تعریف کنیم.

اول اومدیم توسط NLogBuilder، nlog.config رو معرفی کردیم. و میره داخل پوشه bin و دنبال این پوشه میگرده. و توسط متد GetCurrentClassLogger میایم یک لاگر برای این فایل میگیرم. بعد یک try catch نوشتیم و قبل از اینکه هاست بیلدر صدا زده بشه اگر چیزی نیاز داشتیم لاگ میکنیم. اگر این هاست بیلدر به هر دلیلی خطا بده وارد کش میشه و توسط logger.error لاگ میشه و کرش میکنه و خطا رو به ما نشون میده.

و نهایتا nlog رو shutdown میکنیم و تمامی منابعی که در اختیار nlog بوده رو Dispse میکنه.