Unity

public static class UnityConfig

{

public static void RegisterComponents()

{

var container = BuildUnityContainer.Container;

// register all your components with the container here

// it is NOT necessary to register your controllers

//container.RegisterType<IExceptionHandler, ApiExceptionHandler>();

//container.RegisterType<IExceptionLogger, ApiExceptionLogger>();

container.RegisterType<IApiHttpResponseException, ApiHttpResponseException>(new ContainerControlledLifetimeManager());

container.RegisterTypes(

AllClasses.FromLoadedAssemblies(),WithMappings.FromMatchingInterface, WithName.Default, WithLifetime.Transient );

GlobalConfiguration.Configuration.DependencyResolver = new UnityDependencyResolver(container);

}

}

public static class BuildUnityContainer

{

static BuildUnityContainer()

{

RegisterComponents();

}

public static UnityContainer Container {

get;

private set;

}

public static void RegisterComponents()

{

if (Container == null)

{

Container = new UnityContainer();

}

}

}

public static void Register(HttpConfiguration config)

{

// Web API configuration and services

string allowedDomains = System.Configuration.ConfigurationManager.AppSettings["AllowedDomains"];

// When ever you want to consume update your client appliction url as part of Web.Config AllowedDomain Key.

// Like <add key="AlloweDomains" value="http://localhost:3000"/>

/\*

var cors = new EnableCorsAttribute(origins: allowedDomains, headers: "\*", methods: "\*");

cors.SupportsCredentials = true;

config.EnableCors(cors);

\*/

config.EnableCors();

// Web API routes

config.MapHttpAttributeRoutes();

var jsonformatter = GlobalConfiguration.Configuration.Formatters.JsonFormatter;

jsonformatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

// Register Dependecy resolver

UnityConfig.RegisterComponents();

config.Services.Add(typeof(IExceptionLogger), new ApiExceptionLogger()); //Global Logger removed

// config.Services.Replace(typeof(IExceptionHandler), new ApiExceptionHandler());

config.Filters.Add(new ApiExceptionFilter());

GlobalConfiguration.Configuration.Filters.Add(new CustomAuthenticationFilterAttribute());

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

public class CustomAuthenticationFilterAttribute : Attribute, IAuthenticationFilter

{

public async Task AuthenticateAsync(HttpAuthenticationContext context, CancellationToken cancellationToken)

{

await Task.Run(() =>

{

IPrincipal incomingPrincipal = context.ActionContext.RequestContext.Principal;

//IPrincipal genericPrincipal = new GenericPrincipal(new GenericIdentity("Andras", "CustomIdentification"), new string[] { "Admin", "PowerUser" });

//context.Principal = genericPrincipal;

SecurityLogic securityLogic = new SecurityLogic();

string userName = incomingPrincipal.Identity.Name;

bool isActiveUser = securityLogic.isActiveUser(userName);

if (!isActiveUser)

{

context.ErrorResult = new AuthenticationFailureResult("Invalid user", null);

//Debug.WriteLine(String.Format("Incoming principal in custom auth filter AuthenticateAsync method is not authenticated: {0}", incomingPrincipal.Identity.IsAuthenticated));

}

});

}

public async Task ChallengeAsync(HttpAuthenticationChallengeContext context, CancellationToken cancellationToken)

{

await Task.Run(() =>

{

IPrincipal incomingPrincipal = context.ActionContext.RequestContext.Principal;

//Debug.WriteLine(String.Format("Incoming principal in custom auth filter ChallengeAsync method is authenticated: {0}", incomingPrincipal.Identity.IsAuthenticated));

});

}

public bool AllowMultiple

{

get { return false; }

}

}

public class AuthenticationFailureResult : IHttpActionResult

{

public AuthenticationFailureResult(string reasonPhrase, HttpRequestMessage request)

{

ReasonPhrase = reasonPhrase;

Request = request;

}

public string ReasonPhrase { get; private set; }

public HttpRequestMessage Request { get; private set; }

public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)

{

return Task.FromResult(Execute());

}

private HttpResponseMessage Execute()

{

HttpResponseMessage response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

response.RequestMessage = Request;

response.ReasonPhrase = ReasonPhrase;

return response;

}

}

public class ApiExceptionFilter : ExceptionFilterAttribute

{

public override void OnException(HttpActionExecutedContext actionExecutedContext)

{

IExceptionHandlerLogic exceptionHandlerLogic = BuildUnityContainer.Container.Resolve<IExceptionHandlerLogic>();

var errorCode = exceptionHandlerLogic.LogExceptionDetals(actionExecutedContext.Exception, Thread.CurrentPrincipal.Identity.Name, actionExecutedContext.Exception.HelpLink);

Error err = new Error

{

Message = ExceptionMessages.SystemErrorOccured,

MessageDetail = ExceptionMessages.CUSTOM\_ERROR\_MESSAGE\_WITHERRORCODE + errorCode

};

throw new HttpResponseException(actionExecutedContext.Request.CreateResponse(HttpStatusCode.InternalServerError, err));

}

}

public class ApiExceptionLogger : ExceptionLogger

{

public override void LogCore(ExceptionLoggerContext context)

{

IExceptionHandlerLogic exceptionHandlerLogic = BuildUnityContainer.Container.Resolve<IExceptionHandlerLogic>();

var errorCode = exceptionHandlerLogic.LogExceptionDetals(context.ExceptionContext.Exception, Thread.CurrentPrincipal.Identity.Name,context.ExceptionContext.Request.RequestUri.ToString());

Error err = new Error

{

Message = ExceptionMessages.SystemErrorOccured,

MessageDetail = ExceptionMessages.CUSTOM\_ERROR\_MESSAGE\_WITHERRORCODE + errorCode

};

throw new HttpResponseException(context.Request.CreateResponse(HttpStatusCode.InternalServerError, err));

}

}

public class ApiExceptionHandler : ExceptionHandler

{

public override void HandleCore(ExceptionHandlerContext context)

{

string supportEmail = System.Configuration.ConfigurationManager.AppSettings["SupportEmail"];

if (!string.IsNullOrEmpty(supportEmail))

{

supportEmail = " at " + supportEmail;

}

else

{

supportEmail = "";

}

context.Result = new ApiErrorResult

{

Request = context.ExceptionContext.Request,

Content = ExceptionMessages.CUSTOM\_ERROR\_MESSAGE + supportEmail

};

}

private class ApiErrorResult : IHttpActionResult

{

public HttpRequestMessage Request { get; set; }

public string Content { get; set; }

public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)

{

Error err = new Error

{

Message = ExceptionMessages.SystemErrorOccured,

MessageDetail = Content

};

var response = Request.CreateResponse( HttpStatusCode.InternalServerError, err);

return Task.FromResult(response);

}

}

}

public class ApiHttpResponseException : IApiHttpResponseException

{

public IHttpActionResult ThrowResponseException(HttpRequestMessage request, Exception ex, HttpStatusCode statusCode, string message, bool shouldLog)

{

if (shouldLog)

{

IExceptionHandlerLogic exceptionHandlerLogic = BuildUnityContainer.Container.Resolve<IExceptionHandlerLogic>();

var errorCode = exceptionHandlerLogic.LogExceptionDetals(ex, Thread.CurrentPrincipal.Identity.Name, ex.HelpLink);

message = message + ", Reference Error Code : " + errorCode;

}

string messageDetail = string.Empty;

if (ex.Data["ErrorDetail"] != null)

{

messageDetail = ex.Data["ErrorDetail"].ToString();

}

Error err = new Error

{

Message = message,

MessageDetail = messageDetail

};

var errorResponse = request.CreateResponse(statusCode, err);

throw new HttpResponseException(errorResponse);

}

public IHttpActionResult ThrowResponseException(HttpRequestMessage request, Exception ex, HttpStatusCode statusCode, string message)

{

return ThrowResponseException(request, ex, statusCode, message, false);

}

}

[DataContract(Namespace ="")]

public class Error

{

[DataMember]

public string Message { get; set; }

[DataMember(EmitDefaultValue = false)]

public string MessageDetail { get; set; }

}

public class ExceptionHandler : IExceptionHandler

{

public virtual Task HandleAsync(ExceptionHandlerContext context,

CancellationToken cancellationToken)

{

if (!ShouldHandle(context))

{

return Task.FromResult(0);

}

return HandleAsyncCore(context, cancellationToken);

}

public virtual Task HandleAsyncCore(ExceptionHandlerContext context,

CancellationToken cancellationToken)

{

HandleCore(context);

return Task.FromResult(0);

}

public virtual void HandleCore(ExceptionHandlerContext context)

{

}

public virtual bool ShouldHandle(ExceptionHandlerContext context)

{

// return context.ExceptionContext.IsOutermostCatchBlock;

return true;

}

}

public class ExceptionLogger : IExceptionLogger

{

public object LoggedByKey { get; private set; }

public virtual Task LogAsync(ExceptionLoggerContext context,

CancellationToken cancellationToken)

{

if (!ShouldLog(context))

{

return Task.FromResult(0);

}

return LogAsyncCore(context, cancellationToken);

}

public virtual Task LogAsyncCore(ExceptionLoggerContext context,

CancellationToken cancellationToken)

{

LogCore(context);

return Task.FromResult(0);

}

public virtual void LogCore(ExceptionLoggerContext context)

{

}

public virtual bool ShouldLog(ExceptionLoggerContext context)

{

LoggedByKey = "MS\_LoggedBy";

IDictionary exceptionData = context.ExceptionContext.Exception.Data;

if (!exceptionData.Contains("MS\_LoggedBy"))

{

exceptionData.Add("MS\_LoggedBy", new List<object>());

}

ICollection<object> loggedBy = ((ICollection<object>)exceptionData[LoggedByKey]);

if (!loggedBy.Contains(this))

{

loggedBy.Add(this);

return true;

}

else

{

return false;

}

}

}

public interface IApiHttpResponseException

{

IHttpActionResult ThrowResponseException(HttpRequestMessage request, Exception ex, HttpStatusCode statusCode, string message);

IHttpActionResult ThrowResponseException(HttpRequestMessage request, Exception ex, HttpStatusCode statusCode, string message, bool shouldLog);

}