# Global Error Handling in .NET Core Web API

The exception handling features help us dealing with the unforeseen errors which could appear in our code. To handle exceptions we can use the try-catch block in our code as well as finally keyword to clean up resources afterward.

Even though there is nothing wrong with using try-catch blocks in our Actions in Web API project, we can extract all that exception handling into a single centralized place. By doing that, we make our actions more readable and error handling maintainable.

In this article, we are going to handle errors by using a try-catch block first and then to rewrite that by using built-in middleware and our custom middleware for global error handling to show the benefits of this approach.

## Error Handling with Try-Catch Block

To start this example, let’s open the Values Controller from the starting project (Global-Error-Handling-Start project). We are going to find a single Get() method and injected Logger service.

It is a common practice to include the log messages while handling errors, therefore we have created the LoggerManager service. It will log all the messages in C drive, so if you don’t want that behavior, just modify the path in the nlog.config file. For more information about how to use Nlog in .NET Core, you can visit <https://code-maze.com/net-core-web-development-part3/>.

Now, let's modify our action method to return some result and log some messages:

using System;

using LoggerService;

using Microsoft.AspNetCore.Mvc;

namespace GlobalErrorHandling.Controllers

{

[Route("api/[controller]")]

public class ValuesController : Controller

{

private ILoggerManager \_logger;

public ValuesController(ILoggerManager logger)

{

\_logger = logger;

}

[HttpGet]

public IActionResult Get()

{

try

{

\_logger.LogInfo("Fetching all the Students from the storage");

var students = DataManager.GetAllStudents(); //simulation for the data base access

\_logger.LogInfo($"Returning {students.Count} students.");

return Ok(students);

}

catch (Exception ex)

{

\_logger.LogError($"Something went wrong: {ex}");

return StatusCode(500, "Internal server error");

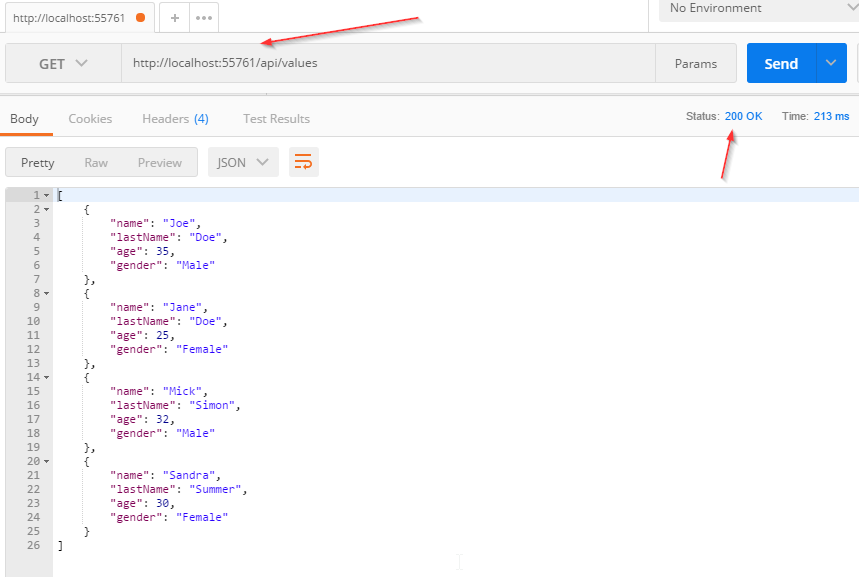
}

}

}

}

When we send a request toward this endpoint, we will get this result:



And the log messages:

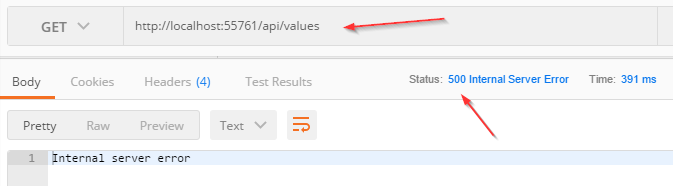


We see that all is working properly.

Now let’s modify our code to force exception by adding the code:

throw new Exception("Exception while fetching all the students from the storage.");

right below the GetAllStudents() method call:



And the log messages:



So, all this works just fine. But the downside of this approach is that we need to repeat our try-catch blocks in all the actions in which we want to catch unhandled exceptions. Well, there is a better approach.

## Handling Errors Globaly with the Built-In Middleware

The UseExceptionHandler middleware is a built-in middleware that we can use to handle exceptions. So, let’s dive to the code to see this middleware in action.

First, we are going to add a new class ErrorDetails in the Models folder:

using Newtonsoft.Json;

namespace GlobalErrorHandling.Models

{

public class ErrorDetails

{

public int StatusCode { get; set; }

public string Message { get; set; }

public override string ToString()

{

return JsonConvert.SerializeObject(this);

}

}

}

We are going to use this class for the details of our error message.

To continue, let's create a new folder Extensions and inside a new static class ExceptionMiddlewareExtensions.cs. Now, we need to modify it:

using GlobalErrorHandling.Models;

using LoggerService;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Diagnostics;

using Microsoft.AspNetCore.Http;

using System.Net;

namespace GlobalErrorHandling.Extensions

{

public static class ExceptionMiddlewareExtensions

{

public static void ConfigureExceptionHandler(this IApplicationBuilder app, ILoggerManager logger)

{

app.UseExceptionHandler(appError =>

{

appError.Run(async context =>

{

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

context.Response.ContentType = "application/json";

var contextFeature = context.Features.Get<IExceptionHandlerFeature>();

if(contextFeature != null)

{

logger.LogError($"Something went wrong: {contextFeature.Error}");

await context.Response.WriteAsync(new ErrorDetails()

{

StatusCode = context.Response.StatusCode,

Message = "Internal Server Error."

}.ToString());

}

});

});

}

}

}

In the code above, we create an extension method in which we register the UseExceptionHandler middleware. Populate the status code and the content type of our response, log the error message and finally return the response with the custom created object.

To be able to use this extension method, let’s modify the Configure method inside the Startup class:

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerManager logger)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.ConfigureExceptionHandler(logger);

app.UseMvc();

}

Finally let’s remove the try-catch block from our code:

[HttpGet]

public IActionResult Get()

{

\_logger.LogInfo("Fetching all the Students from the storage");

var students = DataManager.GetAllStudents(); //simulation for the data base access

throw new Exception("Exception while fetching all the students from the storage.");

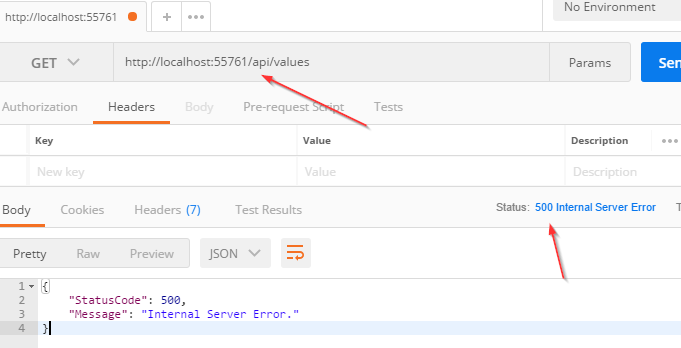
\_logger.LogInfo($"Returning {students.Count} students.");

return Ok(students);

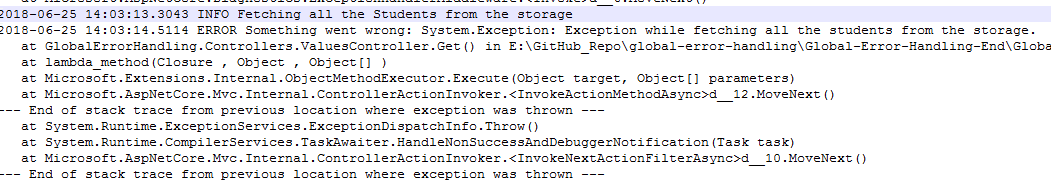
}

And there you go. Our action method is much cleaner now and what’s more important all of our future actions will be more readable as well, without try-catch blocks.

So let’s inspect the result:



And the log messages:



Excellent.

Now, we are going to show you how to use custom middleware for global error handling.

## Handling Errors Globaly with the Custom Middleware

Let’s create a new folder CustomExceptionMiddleware and inside a new class ExceptionMiddleware.

We are going to modify that class:

using GlobalErrorHandling.Models;

using LoggerService;

using Microsoft.AspNetCore.Http;

using System;

using System.Net;

using System.Threading.Tasks;

namespace GlobalErrorHandling.CustomExceptionMiddleware

{

public class ExceptionMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILoggerManager \_logger;

public ExceptionMiddleware(RequestDelegate next, ILoggerManager logger)

{

\_logger = logger;

\_next = next;

}

public async Task InvokeAsync(HttpContext httpContext)

{

try

{

await \_next(httpContext);

}

catch (Exception ex)

{

\_logger.LogError($"Something went wrong: {ex}");

await HandleExceptionAsync(httpContext, ex);

}

}

private static Task HandleExceptionAsync(HttpContext context, Exception exception)

{

context.Response.ContentType = "application/json";

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

return context.Response.WriteAsync(new ErrorDetails()

{

StatusCode = context.Response.StatusCode,

Message = "Internal Server Error from the custom middleware."

}.ToString());

}

}

}

In the code above, the first thing we do is register our IloggerManager service and RequestDelegate through the dependency injection. The \_next parameter of RequestDeleagate type is a function delegate which can process our HTTP requests.

After the registration process, we create the InvokeAsync() method. It is obligatory for the RequestDelegate to be able to process requests. We can also use Invoke() method instead if we want to process request synchronously.

If everything goes well, the \_next delegate will process the request and the Get action from our controller will generate the successful response. But if a request is unsuccessful (and it is, because we are forcing exception), our middleware will trigger the catch block and call the HandleExceptionAsync method.

In that method, we just set up the response status code and content type and return a response.

Now let’s modify our ExceptionMiddlewareExtensions class with another static method:

public static void ConfigureCustomExceptionMiddleware(this IApplicationBuilder app)

{

app.UseMiddleware<ExceptionMiddleware>();

}

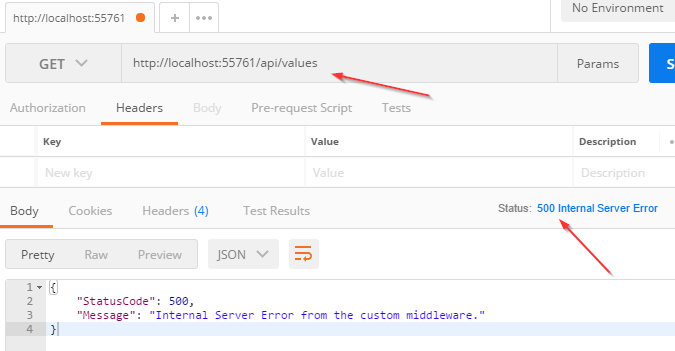
Finally, let’s use this method in the Configure method in the Startup class:

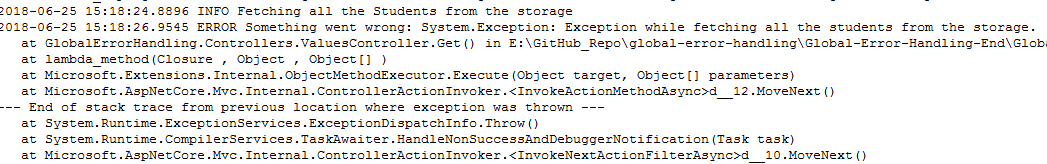
//app.ConfigureExceptionHandler(logger);

app.ConfigureCustomExceptionMiddleware();

Great.

Now is the time to inspect the result:





## Conclusion....