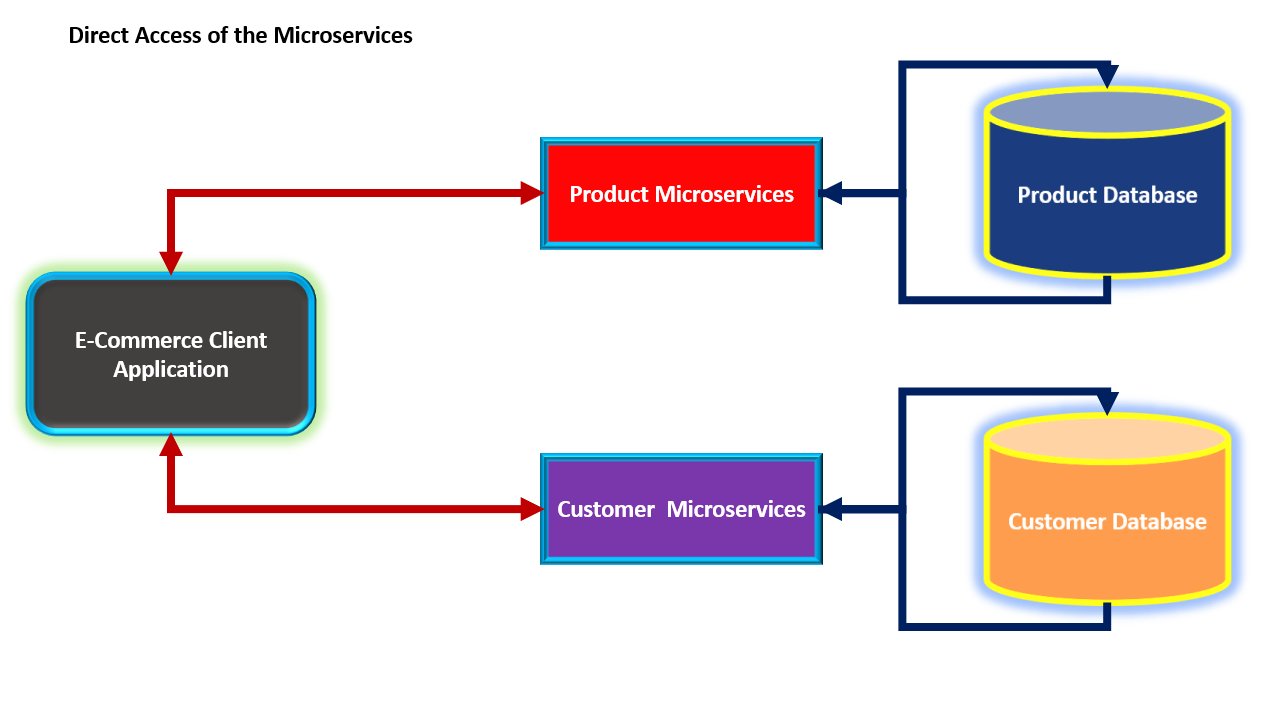
**-**[**July 17, 2023**](https://www.webnethelper.com/2023/07/aspnet-core-creating-resilient.html)

**In this article,  we will see how to implement resilient Microservices using ASP.NET Core. The question here is why we need to implement such a design for the Microservices? Consider, the design for Microservices shown in Figure 1**

**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjnNGv1dvQ-yhF7cjGamutB4Hl3pgUFpWR5WTb40XsuUlgJsgAPtAZ5wcH88RYjfYRX5MqoOlXUdiKOrBiizAP-UCw7mK79TObBO38C_wu2eMsVI3XgcQHKl8dwIZ-_cUA4PBdKnUoPRL2GAVdiHU2se44xozHAy2F_8joS1boDxvGux5lwZi5p10Cd-7k/s1286/Figure-1-Direct-Access-ForMicroservices.png)**

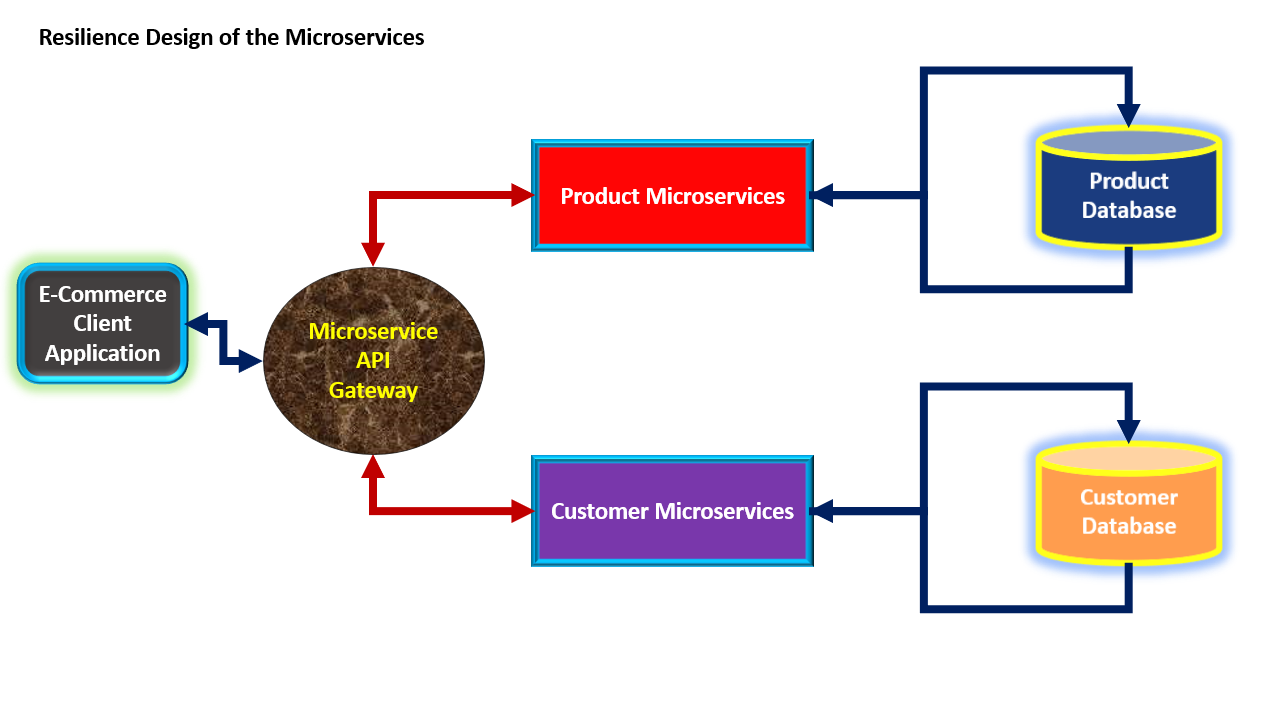
**Figure 1: The Direct Access of Microservices from the client application**

**As shown in Figure 1, the E-Commerce client application directly accesses Product and Customer Microservices. The client application will be able to receive data from these services successfully if the Product and the Customer services are up and running. But in case one of these services is down because of some errors, then the client application will not get the data and hence might be a reason to crash the client app. In this case, although the client app is not responsible for the stability of the Microservices still any issue that occurs in Microservices will crash the client application.**

**The question here is, how to resolve such kind of a design flow in our application?**

**The solution here is implementing resiliency in the Microservices design. In the software system (and typically for Microservices applications), "Resiliency" is defined as the ability to maintain acceptable availability for the service so that all dependant components on the service will be able to work stably instead of crashing. To implement resiliency in design we can add a Gateway Controller service which will act as a bridge between the client application and actual Microservices. In such case,**

**if the actual Microservice is down then the Gateway Service will make sure that the fallback response will be given to the client to make sure that the client application will work without any issues. Figure 2 shows an implementation of the resilience design of the application.**

**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjFyVF_2488cQU2ORW1iOa4uqWDk-RAMuClW5DILsdRRq_AKWnCBP6xvMly0qlTDO8sX4JOFTMY2jAdve8v9ZjF9l1tTcilstuLrLaBzgasupzm2vxD_K1QJO1JaXEn3_u-ptnUPTYR1U0xrdDvFmV3gLZzgBZI-yZMQy_VbNZDsN_xJd0X5Q8GC2kbbco/s1282/Figure-2-Resilient-Design.png)**

**Figure 2: The Resilience design**

**As shown in Figure 2, the Microservice API Gateway service is used to act as a proxy between the client app and actual Microservices. This design will make sure that the API Gateway Service will take care of the Increased Load, Network Issues, etc. In this case, if the actual service is unavailable because of a network issue the n API Gateway will take care of communication by handling retry.**

**To implement the resilient design we will be using the Polly library.**[**Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, Rate-limiting, and Fallback in a fluent and thread-safe manner.**](https://github.com/App-vNext/Polly)

**The Application Implementation**

**The application is developed using .NET 7 and Visual Studio 2022 Enterprise Edition. (Please Note that, although in the image I have shown Database,  to focus on the concept I have not used Database.). To implement the code for this projet, you must have an experience in ASP.NET Core.**

**Step 1: Open Visual Studio and create a new ASP.NET API Core application targetted to .NET 7, and name it ProductsService. In the project add a new folder named Models. In this folder add a new class file and name it Product.cs. In this file add code as shown in Listing 1**

**public class Product**

**{**

**public int ProductId { get; set; }**

**public string? ProductName { get; set; }**

**public int Price { get; set; }**

**public string? Url { get; set; }**

**}**

**Listing 1: The Product class**

**In the same folder, add a new class file and name it ProductsDb.cs. In this file, add the code as shown in Listing 2. This code hard-coded product data.**

**public class ProductsDb:List<Product>**

**{**

**public ProductsDb()**

**{**

**Add(new Product() { ProductId = 1, ProductName = "Laptop", Price = 340000 });**

**Add(new Product() { ProductId = 2, ProductName = "Hard Disk", Price = 340000});**

**Add(new Product() { ProductId = 3, ProductName = "RAM", Price = 340000});**

**Add(new Product() { ProductId = 4, ProductName = "Printer", Price = 340000});**

**Add(new Product() { ProductId = 5, ProductName = "SSD", Price = 340000});**

**Add(new Product() { ProductId = 6, ProductName = "USB", Price = 340000});**

**Add(new Product() { ProductId = 7, ProductName = "Laptop Charger", Price = 340000});**

**Add(new Product() { ProductId = 8, ProductName = "DVD ROM", Price = 340000});**

**Add(new Product() { ProductId = 9, ProductName = "Mouse", Price = 340000});**

**Add(new Product() { ProductId = 10, ProductName = "Keyboard", Price = 340000});**

**}**

**}**

**Listing 2: The Products data**

**Step 2: In the project add a new folder and name it as Services. In this Services folder add a new class file and name it as ProductsRepository.cs. In this class file, add the code for returning the List of Products as shown in Listing 3**

**public class ProductsRepository**

**{**

**ProductsDb \_db;**

**public ProductsRepository()**

**{**

**\_db = new ProductsDb();**

**}**

**public List<Product> GetProducts() => \_db;**

**}**

**Listing 3: The ProductsRepsitory**

**Step 3: Modify the Program.cs for registering ProdutsRepository class in the dependency container and also defining JSON response from REST API as shown in Listing 4**

**builder.Services.AddScoped<ProductsRepository>();**

**builder.Services.AddControllers().AddJsonOptions(options => options.JsonSerializerOptions.PropertyNamingPolicy = null);**

**Listing 4: The dependency registration of ProductsRepository and JSON serialization**

**Step 4: In the Controllers folder, add a new Empty API Controller and name it ProductsController.cs. In this controller, we will inject the ProductsRepsitory using constructor injection. The controller contains Http Get method to return the Products data. The code for the controller is shown in Listing 5**

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

**[ApiController]**

**public class ProductsController : ControllerBase**

**{**

**private ProductsRepository \_repository;**

**public ProductsController(ProductsRepository repository)**

**{**

**\_repository = repository;**

**}**

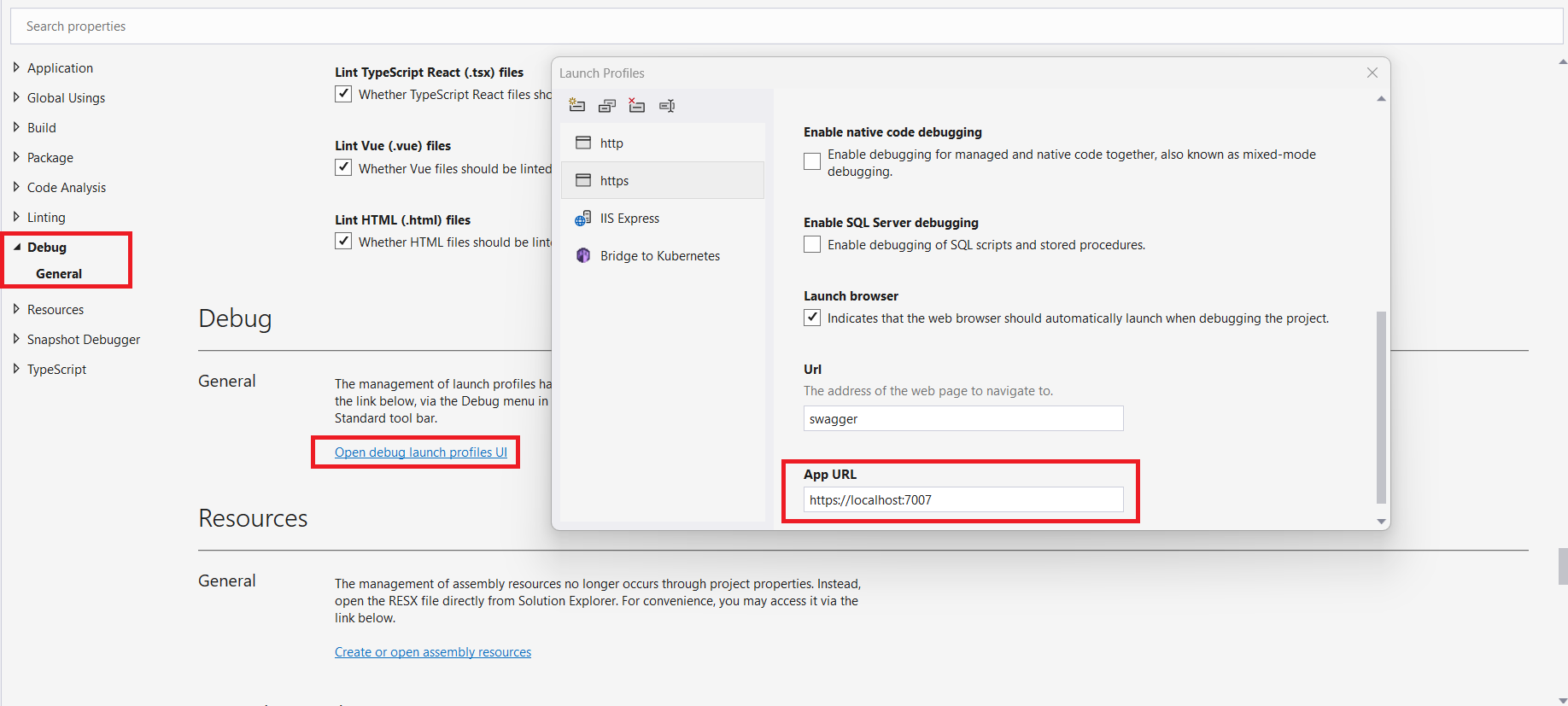
**[HttpGet]**

**public List<Product> Get() => \_repository.GetProducts();**

**}**

**Listing 5: The ProductsController.cs**

**Step 5: We will expose this API on port 7007, we will change the port from the project properties as shown in Figure 3**

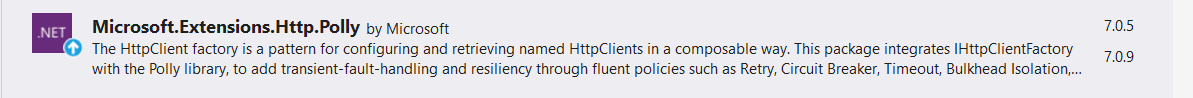
**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEhIevH96lulrfuFcfELQw4lFjilPfkGTEcZWiCONBIOLWcCJMZuuZdXASyubefITML6UzjRvZ44SO5UaxxpeOCCZXckbXvEe3Shlui5f9ly2oyFw-C0YyBl-aFwwdlEAqJfNEs7Hm97GBLBx6MBz8u0TWRbNZVhWuNIHKVYZ9vN_Wmid_CT0bk0h5RTtNU/s1836/Figure-3-Port.png)**

**Figure 3: Setting The Port for the Project**

**Run the project and Test it. Similarly, you can create a CustomerService. (The code for Customer Service can be downloaded from the git link given at the end of the article.)**

**Creating the Resilience Microservices**

**In this part, we will create a resilient service using the Polly library. Let's create a new solution in a new instance of VS 2022 and name it as CircuitBreakerGateway. In this solution, add a new ASP.NET Core API project and name it as CircuitBreakerGateway. In this project, add the Polly library as shown in Figure 4**

**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEha9pQhIDPknaHYMW4sNDmwh7DQZzKDU0-bm_yzvsIv6ZEsI05TR02lK9HD236UMNjR6tPCnoLs2RaODgJGVn9np0noZqkk4dfBm6iLYbcimBwpWzuSO4jLdW6I57fP4G_pCLDkSr2bilIQgAaQhTSsoUeFXR4xid0-lw3QUY8hy7od0RkdxVPVHQl4Yms/s1193/Figure-4-Polly-Library.png)**

**Figure 4: Polly Library**

**Step 6: In this project, in the Controllers folder, add a new Empty API Controller named GatewayController.cs. In this controller, we will inject the IHttpClientFactory type using constructor injection. We need this to access actual Services. The Controller class also uses the following classes to manage Retry and Circuit**

* **AsyncFallbackPolicy:**
  + **This class is used to define the fallback policy to make sure that when the actual service is not available then send a fallback message to the client.**
  + **The fallback response value is sent using the FallbackAsync() method. This is an extension method to PolicyBuilder class.**
* **AsyncRetryPolicy:**
  + **This class is used to define the retry count to access the actual service if it not responding. Once the retry count is over the exception will be thrown.**
  + **The retry cunt is set using the RetryCount() extension method of the PolicyBuilder class.**
* **AsyncCircuitBreakerPolicy:**
  + **This class is used to define the policy for breaking the circuit by using CircuitBReakerAsync() method. to make sure that the circuit will be kept open before resetting it.**
* **AsyncPolicyWrap**
  + **This class is used to wrap polli policies to manage the execution.**
  + **These policies are wrapped and applied using WrapAsync() method.**

**The controller CallTo() method. This method contains code to call actual Services from the Gateway Service. To make sure that the call to the actual service will be made under the Retry and Circuit Breaker policies, the call to the actual service will be wrapped in the ExecuteAsync() method of the AsyncPolicyWrap class. The code for the GatewayController is shown in Listing 6.**

**using Microsoft.AspNetCore.Mvc;**

**using Polly;**

**using Polly.CircuitBreaker;**

**using Polly.Fallback;**

**using Polly.Retry;**

**using Polly.Wrap;**

**namespace CircuitBreakerGateway.Controllers**

**{**

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

**[ApiController]**

**public class GatewayController : ControllerBase**

**{**

**private readonly HttpClient \_httpClient;**

**private readonly AsyncFallbackPolicy<IActionResult> \_asyncFallbackPolicy;**

**private readonly AsyncRetryPolicy<IActionResult> \_asyncRetryPolicy;**

**private static AsyncCircuitBreakerPolicy? \_asyncCircuitBreakerPolicy;**

**private readonly AsyncPolicyWrap<IActionResult> \_asyncPolicy;**

**public GatewayController(IHttpClientFactory factory)**

**{**

**// Defing the policy**

**// to return the fallback response**

**\_asyncFallbackPolicy = Policy<IActionResult>**

**.Handle<Exception>()**

**.FallbackAsync(Content("Sorry, currently we are not able to process your request"));**

**// The retry policy**

**\_asyncRetryPolicy = Policy<IActionResult>**

**.Handle<Exception>()**

**.RetryAsync(2); // Retry 2 times**

**/// The CircuitBreakerAsync() method accepts 2 parameters**

**/// 1. The Number of exceptions allowed before opening circuit**

**/// 2. The duration the circuit will stay open before resetting**

**if (\_asyncCircuitBreakerPolicy == null)**

**{**

**\_asyncCircuitBreakerPolicy = Policy**

**.Handle<Exception>()**

**.CircuitBreakerAsync(2, TimeSpan.FromMinutes(1));**

**}**

**\_asyncPolicy = Policy<IActionResult>**

**.Handle<Exception>()**

**.FallbackAsync(Content("Sorry, currenty we are not able to process your request"))**

**.WrapAsync(\_asyncRetryPolicy)**

**.WrapAsync(\_asyncCircuitBreakerPolicy);**

**\_httpClient = factory.CreateClient();**

**}**

**[HttpGet]**

**[ActionName("products")]**

**public async Task<IActionResult> Products()**

**=> await CallTo("https://localhost:7007/api/Products");**

**[HttpGet]**

**[ActionName("customers")]**

**public async Task<IActionResult> Customers()**

**=> await CallTo("https://localhost:6007/api/Customers");**

**//Method to make a call to the Actual API**

**private async Task<IActionResult> CallTo(string url)**

**=> await \_asyncPolicy.ExecuteAsync(async () => Content(await \_httpClient.GetStringAsync(url)));**

**}**

**}**

**Listing 6: The GatewayController class**

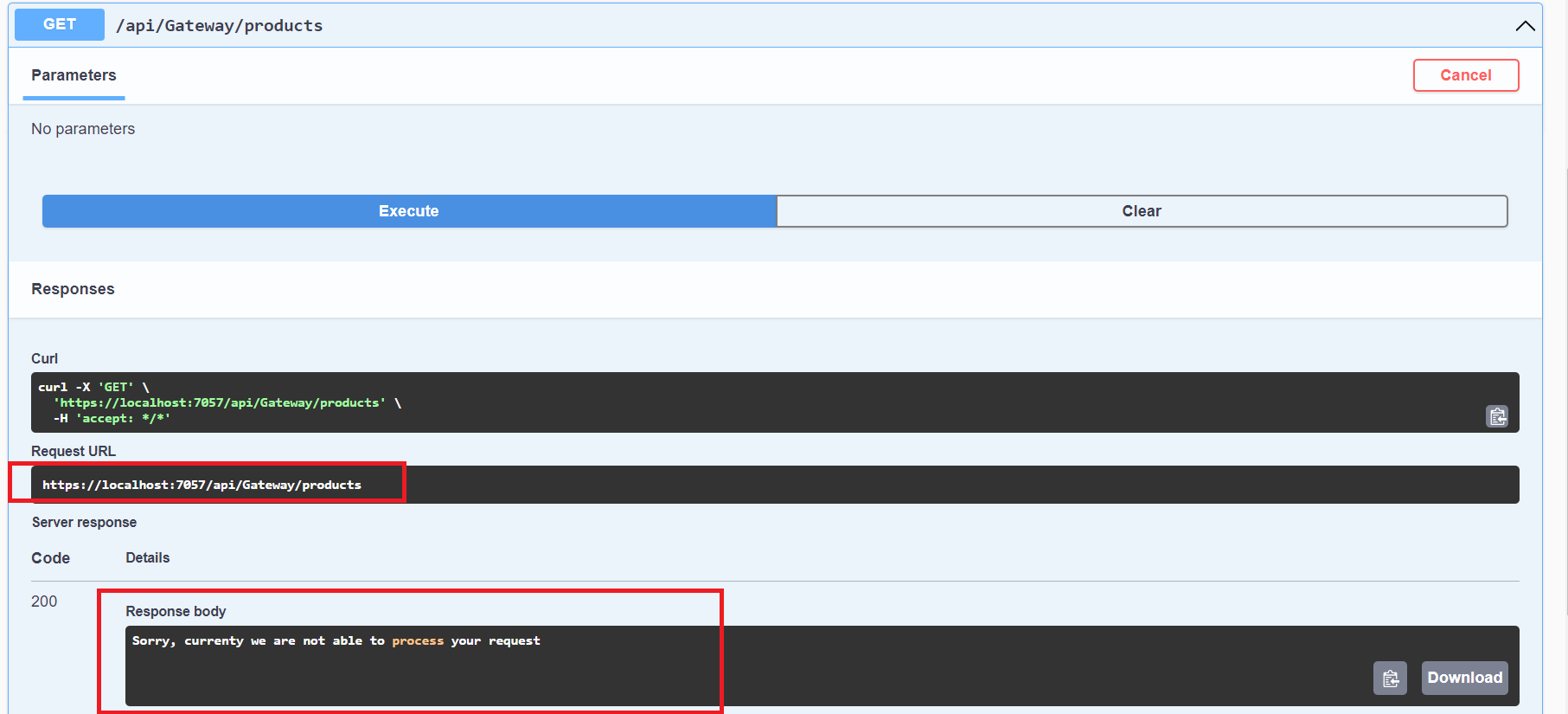
**The GatewayController is the heart of the design. This controller will make sure that if the actual service is down two times retry will be made otherwise fallback will be shown instead of an error and within one minute if the actual service is up then the response from the actual service will be shown.**

**Step 7: Let's resolve the IHttpClientFactory type by calling AddHttpClient() method on the IServiceCollection in Program.cs as shown in Listing 7**

**builder.Services.AddHttpClient();**

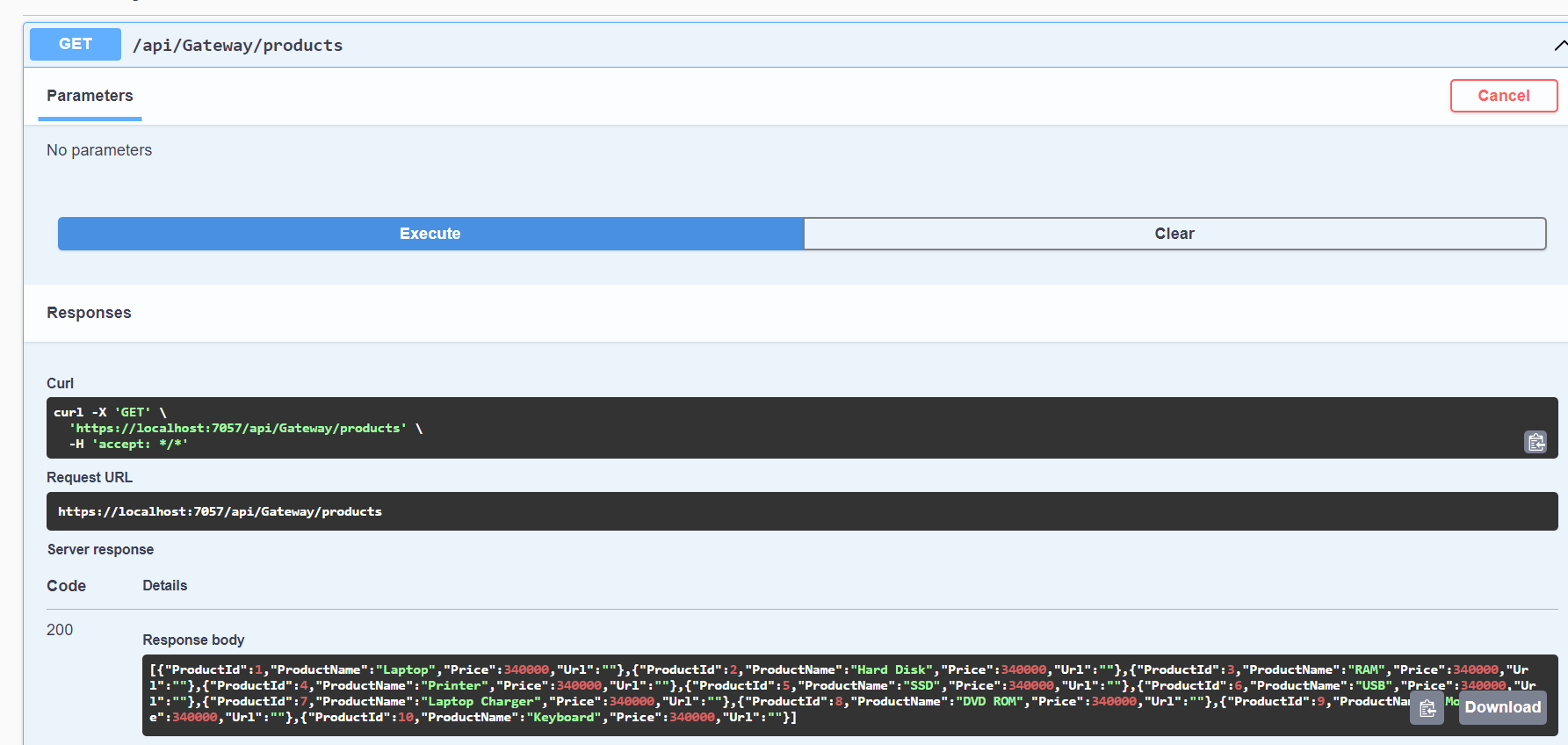
**Listing 7: Registration of the IHttpClientFactory**

**To run Gateway Service, Product, and Category Services, make sure that they are opened in 3 separate VS 2022 instances. Now, run the Gateway service and try to call the Product service from the Swagger UI, the fallback UI will be displayed as shown in Figure 5**

**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEgZ3klGNiacf3vc0OnyOVNw_ivvGe362WvYazcWq2GFp8oueSKo83FckcbpBnUvOLz59fKWZQpWV2fT543lZYnRP0LU_qgzf2xuYQDeJt5T6gC47uEyI_Co1ICaviFv_qW5H86AqUxkz13-FPltinDWSzZsEidjlNu1YjBM1E4SN62km7yJGcvQCPjifyA/s1813/Figure-5-The-Fallback-UI.png)**

**Figure 5: The fallback UI**

**Again make a call to Product Service (or Category Service) using the Gateway service while the call is under process start the Product Service (or Category Service), once the actual service starts, the data will be received and displayed as shown in Figure 6**

**[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEizFn9s_1CRKtXJoQ7WfPLcbiyg98kEPp6GsIpMQ1tVgK3f_N7uPpZExR3o8SvXiguEFNMF4axFMLZMQiZY1GsGjd7Sonv_sK4cV33cJG48UtfUR10_uMpQ2oGbXOtcLqlTlFcNX2ji__d9157kyInDaq7GouhPxdkcuKogCOtPOl9pq04aZu22lWR7Oyw/s1785/Figure-6-The-Product-Service.png)**

**Figure 6: The response from the Product service**

**That's it.**

**The Code for this can be downloaded from**[**this link**](https://github.com/maheshsabnis/resilientmicroservices.git)**.**

**Conclusion: The Resilient design is one of the recommended designs for Modern Microservices based applications. When having multiple services, because of heavy network traffic or the more resource utilization of resources on service the service may not respond immediately so resiliency is the best option to design services.**