University of Mumbai Institute of Distance & Open Learning



Dr.Shankar Dayal Sharama Bhavan, Kalina, Vidanagari, Santacruz (E), Mumbai-400 098.

Certificate

This is to certify that

MR. PRAMOD R. SHARMAApplication ID: **92539**, Seat No: **2500135** from Rizvi College of Arts, Science and Commerce Bandra(W), Mumbai 400 050 has successfully completed all the practical of Paper II titled **CLOUD APPLICATION DEVELOPMENT** for M.sc (IT) Part II in the academic year 2023-2024.

MSc (IT) Co-ordinator, IDOL External Examiner

TABLE OF CONTENTS

1. [Develop an ASP.NET Core MVC based Stateless Web App.](#_bookmark0) 6
2. [Develop a Spring Boot API.](#_bookmark1) 11
3. [Create an ASP.NET Core Web API and configure monitoring](#_bookmark2) 15
   1. [Add NLog and ApplicationInsight Packages](#_bookmark3) 15
   2. [Implement Microsoft.Extensions.Logging.ILogger](#_bookmark4) 15
4. [a.Create an Azure Kubernetes Service Cluster](#_bookmark5) 20
5. [Enable Azure Dev Spaces on an AKS Cluster](#_bookmark6) 25
6. [Configure Visual Studio to Work with an Azure Kubernetes Service Cluster](#_bookmark7) 28
7. [Configure Visual Studio Code to Work with an Azure Kubernetes Service Cluster](#_bookmark8) 32
8. [Deploy Application on AKS](#_bookmark9) 32
   1. [Core Web API](#_bookmark10) 32
   2. [Node.js API](#_bookmark11) 32

[5 Create an AKS cluster](#_bookmark12) 37

1. [from the portal](#_bookmark13) 37
2. [with Azure CLI](#_bookmark14) 37
3. [Create an Application Gateway Using Ocelot and Securing APIs with Azure AD.](#_bookmark15) 40

[Introducing Ocelot](#_bookmark16) 40

[The Order Processing Microservices-Based Application](#_bookmark17) 40

[Prerequisites](#_bookmark18) 40

[The solution structure](#_bookmark19) 40

[Create the projects for the Order Processing application](#_bookmark20) 42

[Create the Customer microservice](#_bookmark21) 42

[Create the CustomerRepository class](#_bookmark22) 42

[Create the CustomerController class](#_bookmark23) 44

[Create the Product microservice](#_bookmark24) 45

[Create the ProductRepository class](#_bookmark25) 46

[Create the ProductController class](#_bookmark26) 48

[Implement the API Gateway Using Ocelot](#_bookmark27) 49

[Install the required package](#_bookmark28) 49

[Implement routing](#_bookmark29) 49

[Run the projects](#_bookmark30) 52

[Implement rate limiting](#_bookmark31) 54

[Implement caching](#_bookmark32) 57

[Implement correlation ID](#_bookmark33) 58

[Conclusion](#_bookmark34) 61

[Aside: Securing ASP.NET Core with Auth0](#_bookmark35) 61

1. [Create a database design for Microservices an application using the database.](#_bookmark36) 63

[8 a. Create an API management service](#_bookmark37) 63

[Create a new service](#_bookmark38) 64

[Go to your API Management instance](#_bookmark39) 65

[In the Azure portal, search for and select API Management services.](#_bookmark40) 65

[8 b. Create an API gateway service](#_bookmark41) 66

1. [Demonstrate](#_bookmark42) 68
   1. [Securing APIs with Azure Active Directory.](#_bookmark43) 68
   2. [Issuing a custom JWT token using a symmetric signing key](#_bookmark44) 69
   3. [Pre-Authentication in Azure API Management](#_bookmark45) 69
   4. [AWS API Gateway Authorizer](#_bookmark46) 69

[Lambda authorizer response format](#_bookmark47) 71

[Lambda function response for format 1.0](#_bookmark48) 71

[Lambda function response for format 2.0](#_bookmark49) 72

[Example Lambda authorizer functions](#_bookmark50) 73

[Identity sources](#_bookmark51) 74

[Caching authorizer responses](#_bookmark52) 78

[Create a Lambda authorizer](#_bookmark53) 79

[Troubleshooting Lambda authorizers](#_bookmark54) 80

1. [Create a serverless API using Azure functions](#_bookmark55) 81
2. [Create an AWS Lambda function](#_bookmark56) 85
3. [Build AWS Lambda with AWS API gateway](#_bookmark57) 93

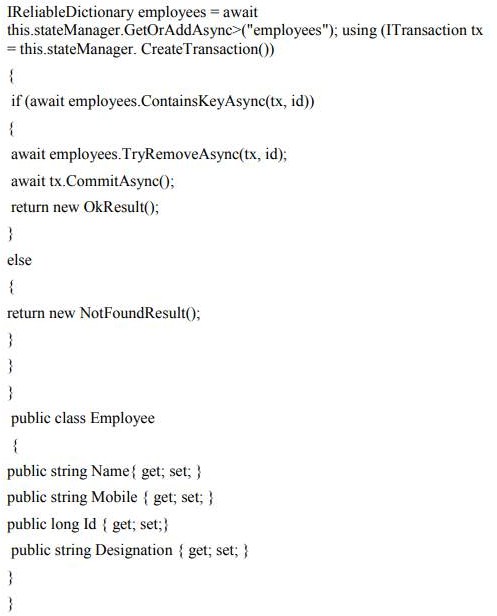
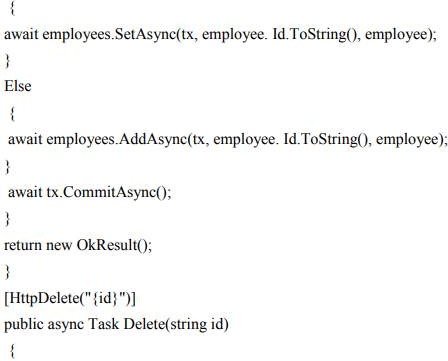
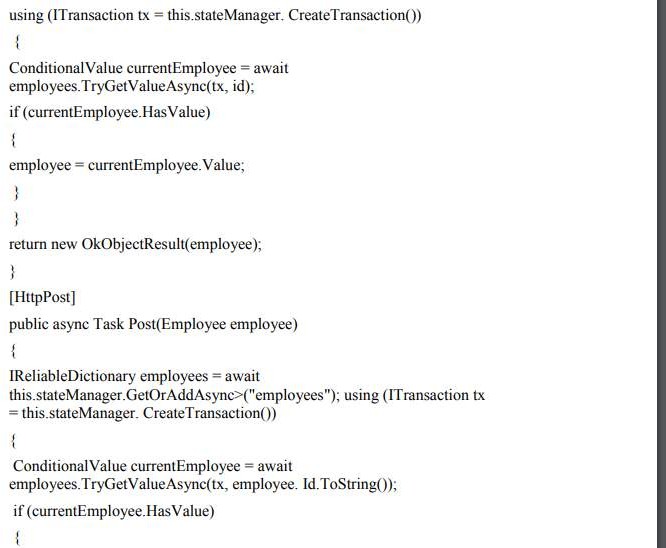
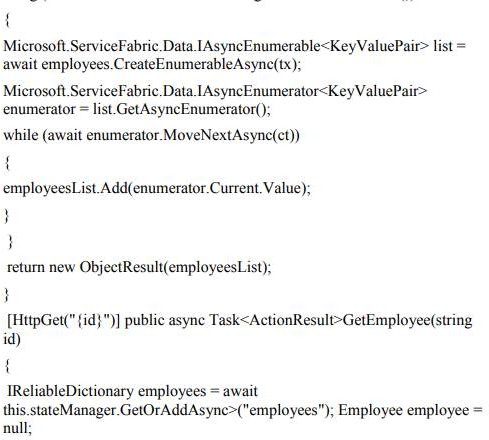
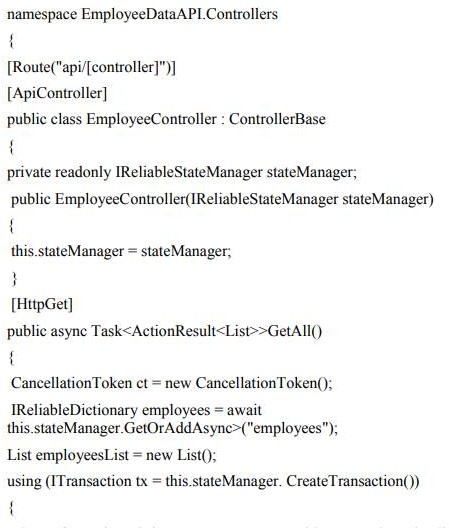
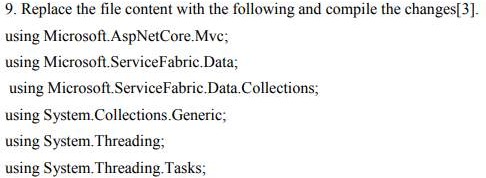
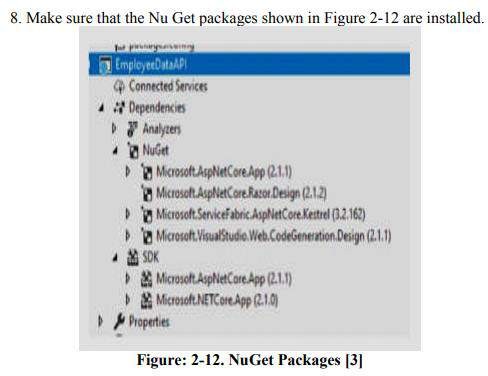
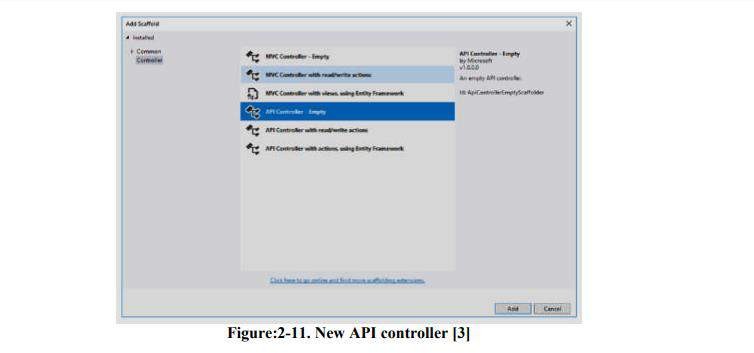
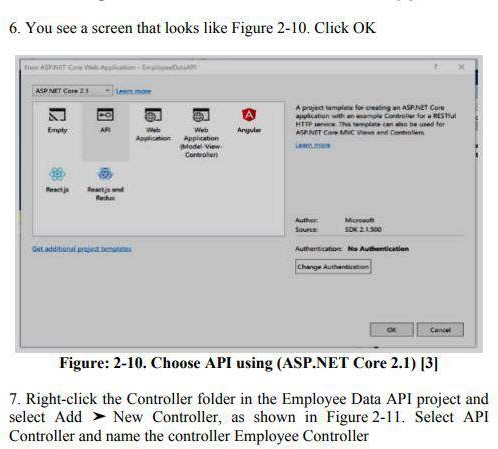
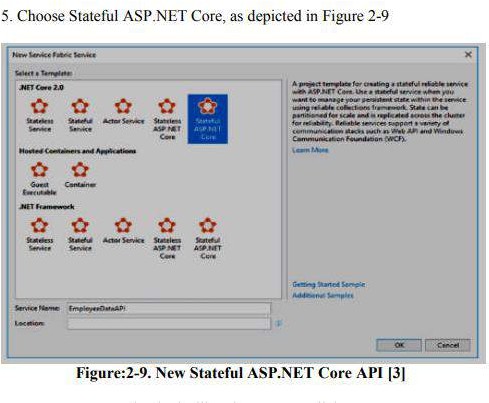
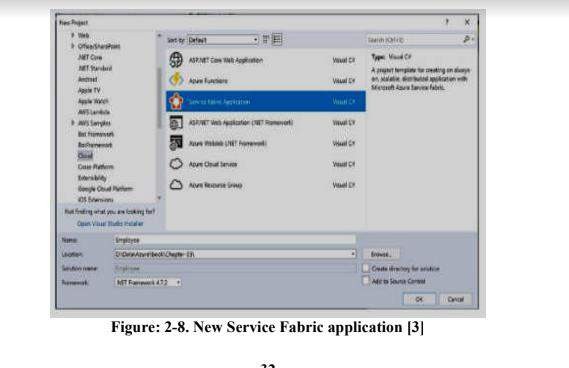
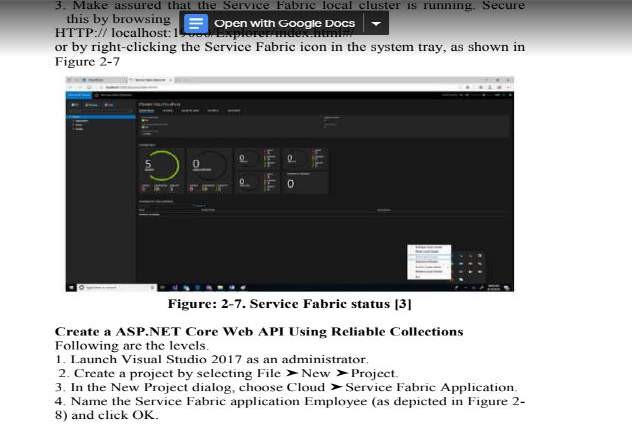
[Step 1: Create a Lambda function](#_bookmark58) 93

[Step 2: Create an HTTP API](#_bookmark59) 94

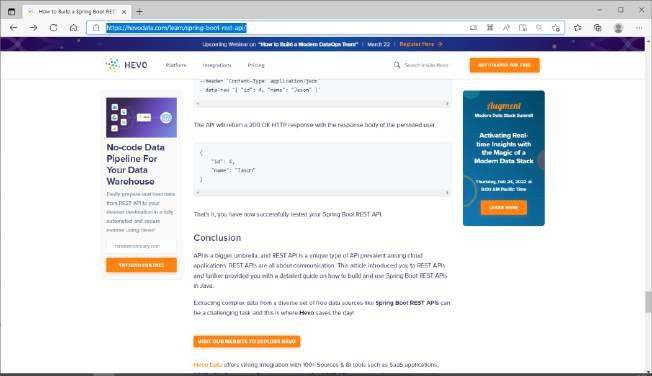
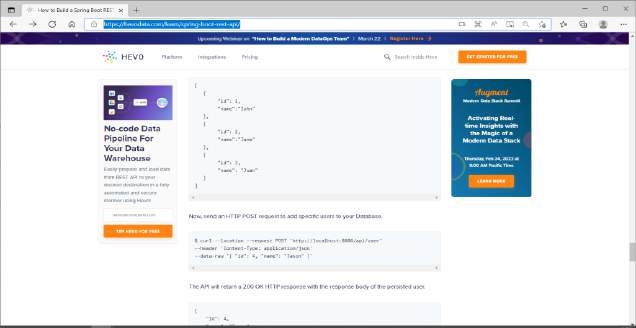
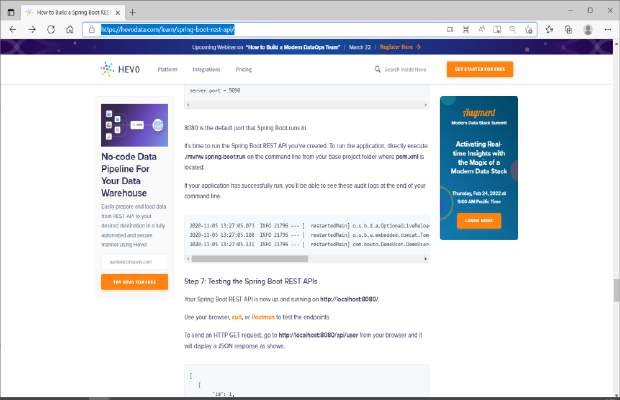
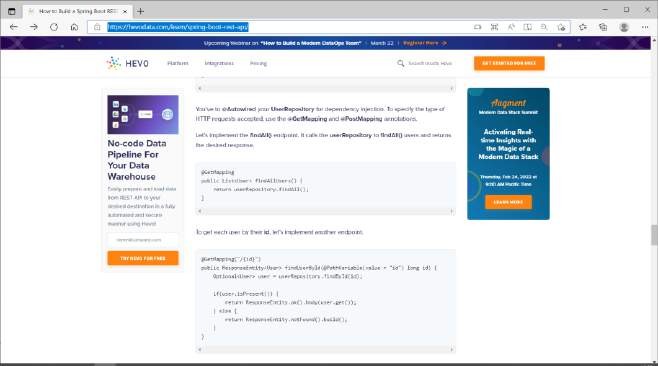
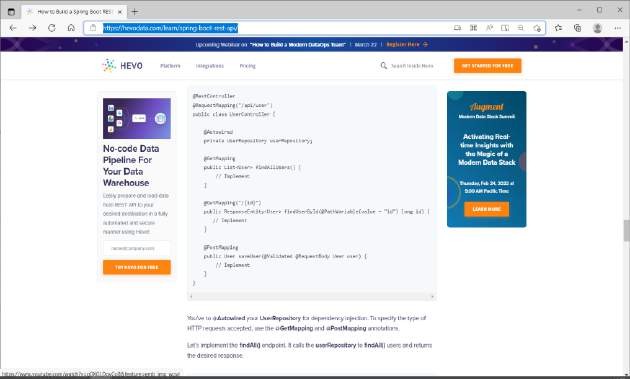
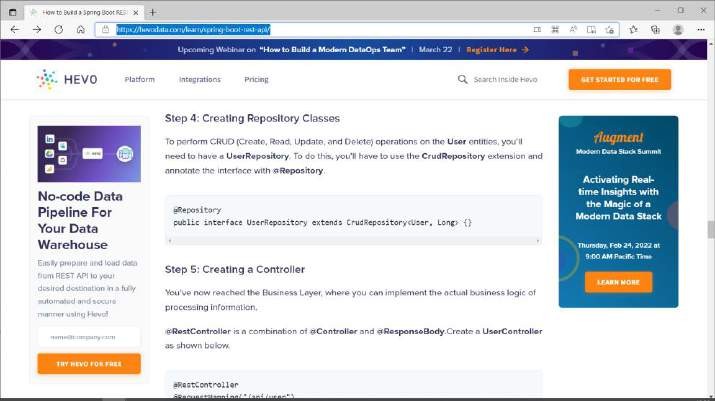
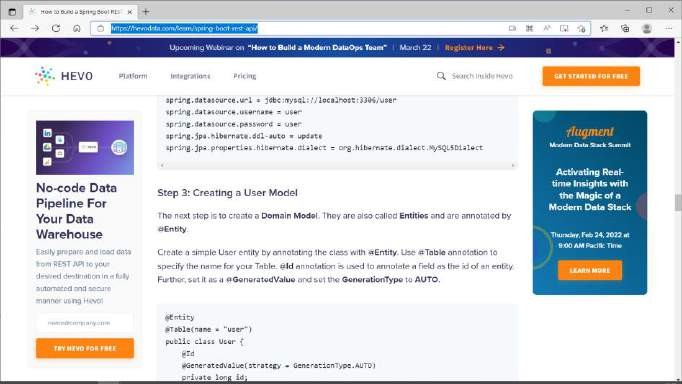
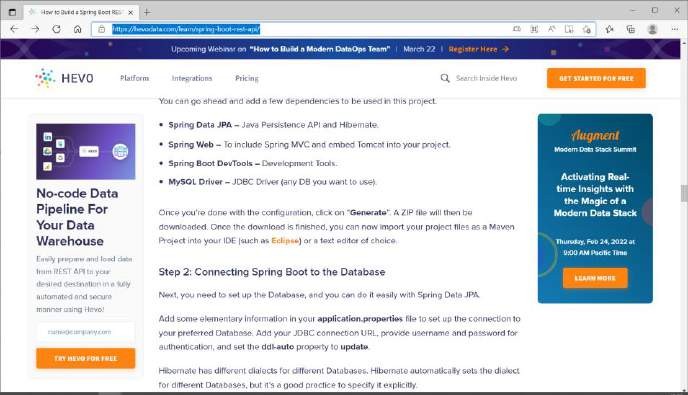
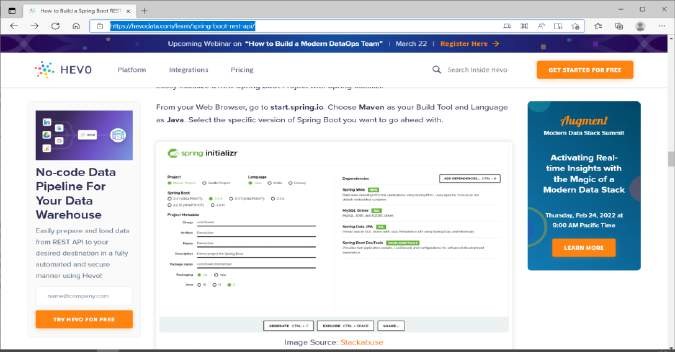
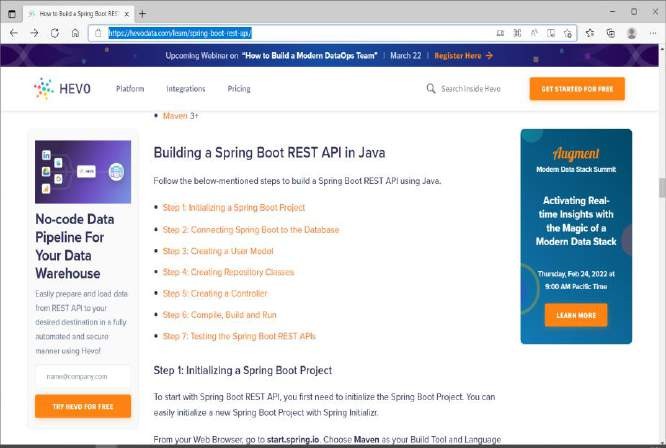
[Step 3: Test your API](#_bookmark60) 95

[(Optional) Step 4: Clean up](#_bookmark61) 96

1. Develop an ASP.NET Core MVC based Stateless Web App.



1. Develop a Spring Boot API.



1. Create an ASP.NET Core Web API and configure monitoring

By the way, this topic belongs to the series to set up an Asp.NET API for production use.

* 1. [API Route Versioning](https://pellerex.com/blog/asp-net-core-web-api-versioning)
  2. [Configuration Management](https://pellerex.com/blog/asp-net-5-web-api-configuration-management)
  3. [Secret Management](https://pellerex.com/blog/asp-net-5-web-api-secret-management)
  4. [Monitoring & Logging (NLog)](https://pellerex.com/blog/asp-net-5-web-api-monitoring)
  5. [Monitoring & Structured Logging (Serilog)](https://pellerex.com/blog/structured-logging-with-serilog)
  6. [Database](https://pellerex.com/blog/database-setup-for-aspnet-5-web-api-with-entity-framework-and-sql)
  7. [Documentation](https://pellerex.com/blog/asp-net-5-api-documentation-with-swagger)
  8. [CORS](https://pellerex.com/blog/asp-net-5-cors)
  9. [Request Validation](https://pellerex.com/blog/asp-net-5-request-validation-with-fluent-validation)
  10. [Global Exception Handling](https://pellerex.com/blog/asp-net-5-global-exception-handling)
  11. [URL Rewriting](https://pellerex.com/blog/url-rewrite-aspnet-core)
  12. [Deploy .NET API to Azure App Service](https://pellerex.com/blog/spa-deploy-aspnet-with-react-using-azure-devops)
  13. Call Other APIs
  14. Distributed Caching
  15. AutoMapper
  16. API Gateways

1. Add NLog and ApplicationInsight Packages

At the end also make sure you have the required packages to support the above implementation:

* + Microsoft.ApplicationInsights.AspNetCore
  + Microsoft.ApplicationInsights.NLogTarget
  + NLog
  + NLog.Web.AspNetCore
  + NLog.WindowsIdentity

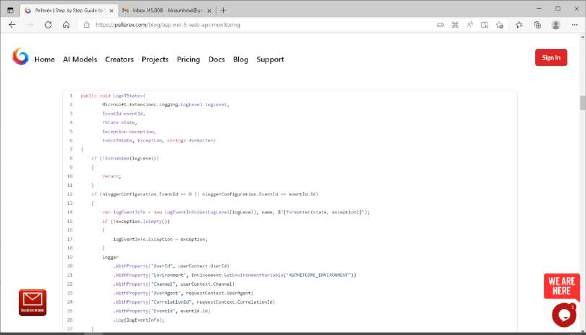
1. Implement Microsoft.Extensions.Logging.ILogger

Starting very simple, and first things first, we need to implement ILogger. As we are using NLog, we will be relying on NLog config files to configure log content and message formats (we will cover nlog.config files in detail later). Our API will be tested and deployed in multiple environment including local machine and production, and hence we define two config files. The reason for that, is on our local machine, we probably don't want the messages to be logged in any Azure Application Insight instance, and hence the log destinations will be different and so are the config files, to keep them separate and clean.

* + nlog.config
  + nlog.production.config

For this reason, when we are implementing the ILogger, we will inject which file to read the configs from, based on the environment. Let's call this method, GetLogConfigFileName.

Next we will need to implement the main Log method, like below:



There are a few points that need to be covered here.

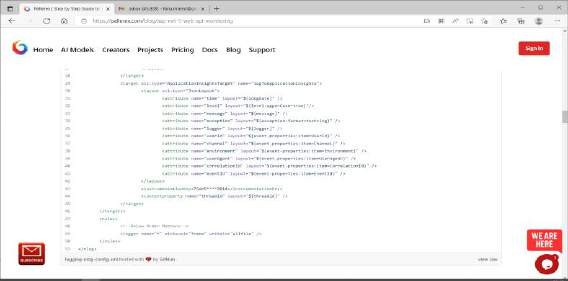
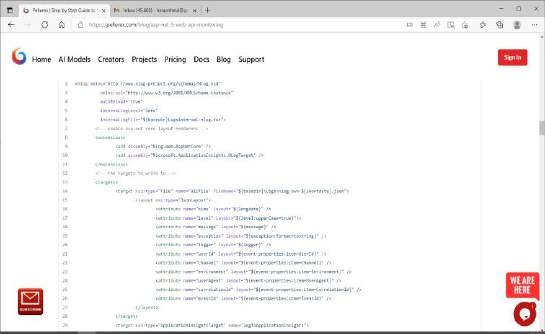
1. **Dynamic Log Messages:** When we log messages or errors, ideally we should have some dynamic information logged along the main message. Parameters such as UserId, Environment, User Agent, etc. These will help us to troubleshoot problems later on when they happen.

**Be mindful of Personally Identifiable Information and Secrets**

You should not log information that would identify information about individual customers such as name, address, date of birth, credentials. However for troubleshooting purposes, we may log UserId, as long as it's a GUID and not an email address. This will in turn help to troubleshoot edge cases, when only a handful of customers have specific errors.

Going back to logging dynamic information with our log messages, you can see we inject those values into log messages using WithProperty method, meaning it will replace the placeholder variable UserId for example, with the actual UserId variable extracted from the http request or user context.

But where did we define these variable and literals? They come from the message format inside nlog.config, like below:



As you can see we have placeholders inside the message such as **event- properties:item=UserId**, which basically says: read the property value UserId as it is supplied using WithProperty method dynamically.

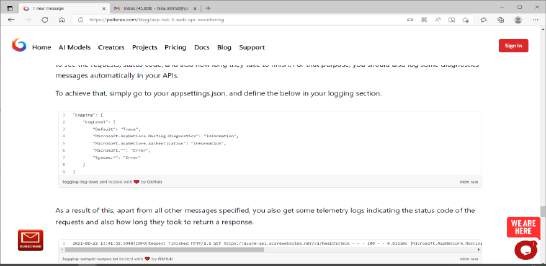
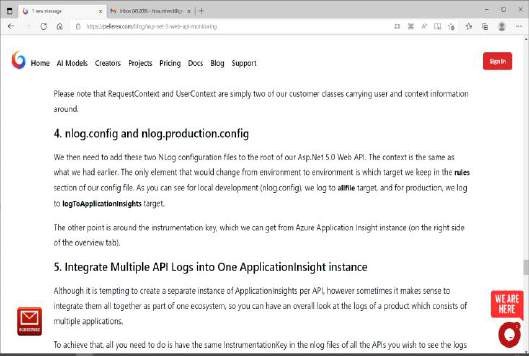
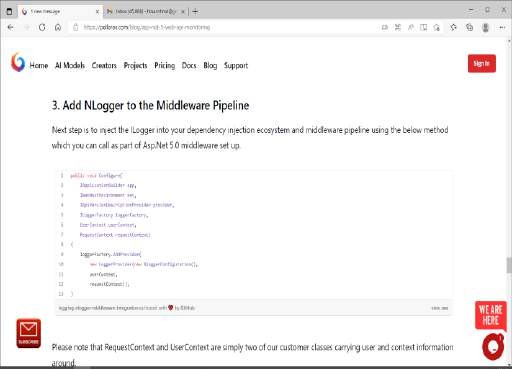
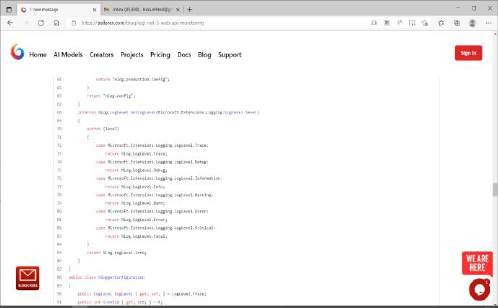
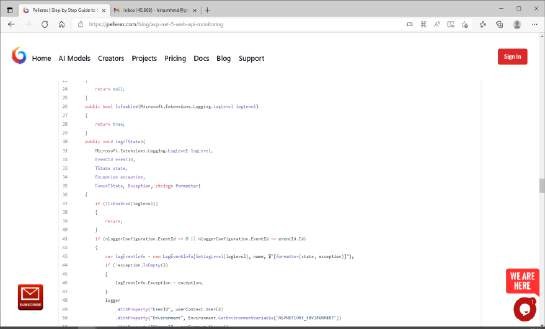
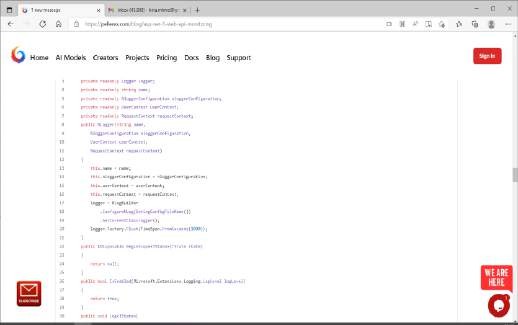
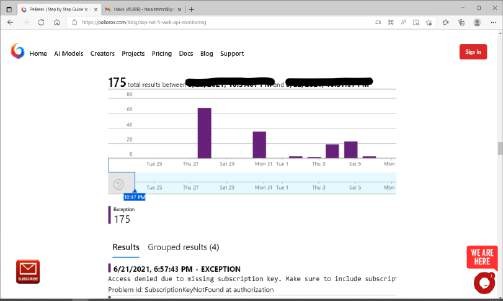
**Structured Logging (aka. Semantic Logging)**

Structured Log format is the ability to separate the log template and log parameters which means logs are not simply converted to text and stored in a text file or a database, which makes it difficult to run query over them. As an example, except if we use some regex to identity certain elements in a pile of text, we won't be easily able to query the average request performance of a specific user over a period of time for an API. With structured logging, this is possible, because we store log records in a structured format, by separating data fields, that in turn enables us to run query over them after storage.

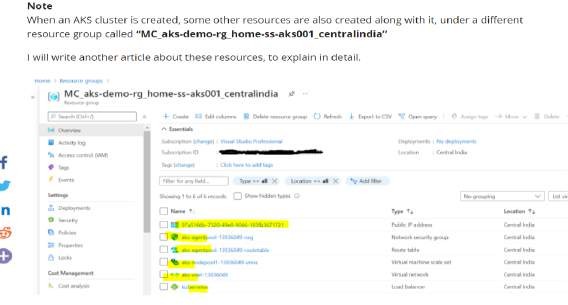
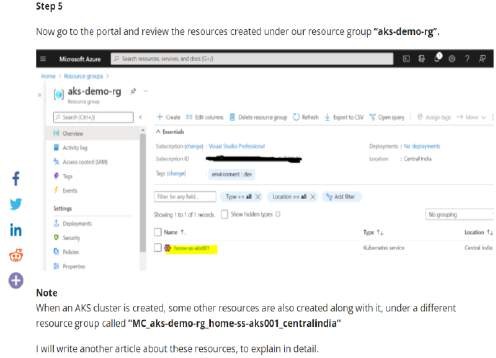
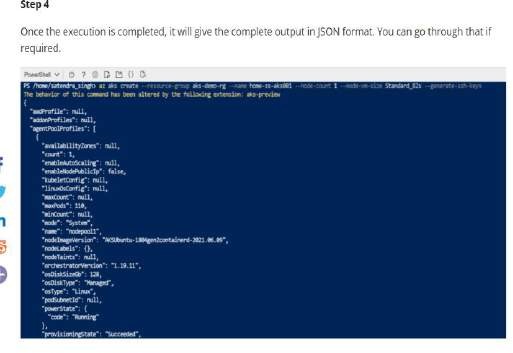
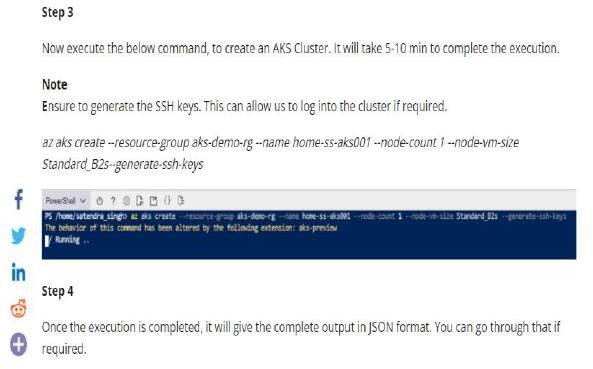
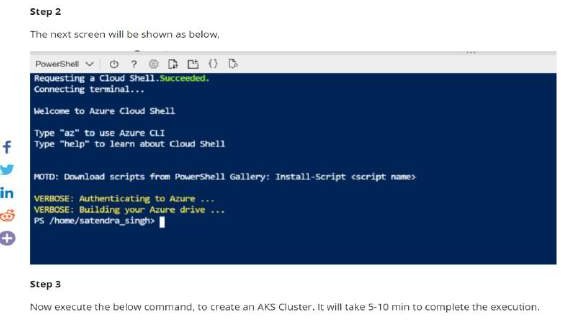
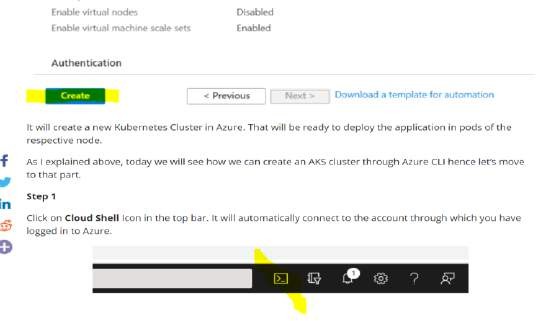
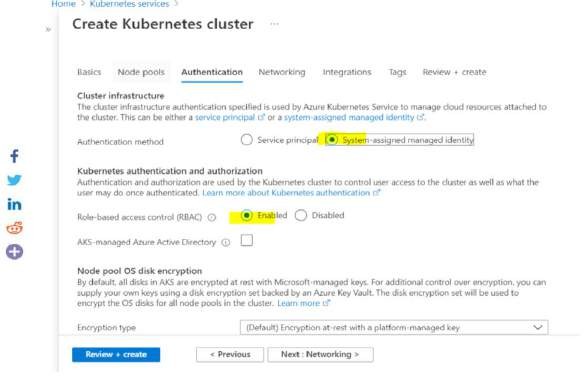
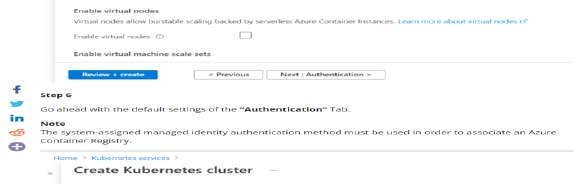
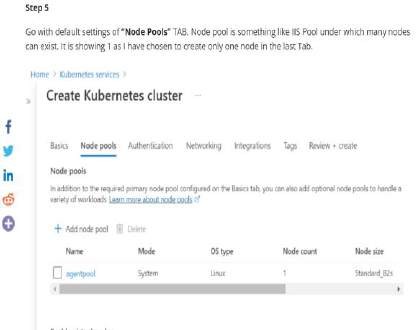
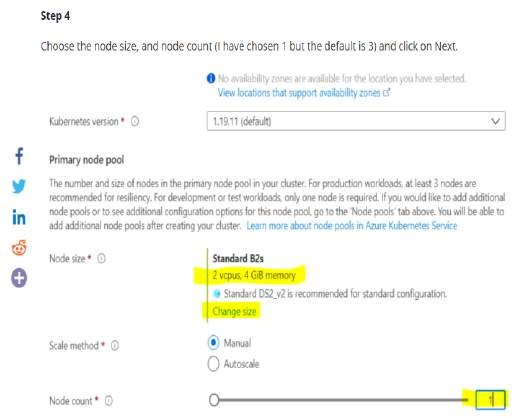
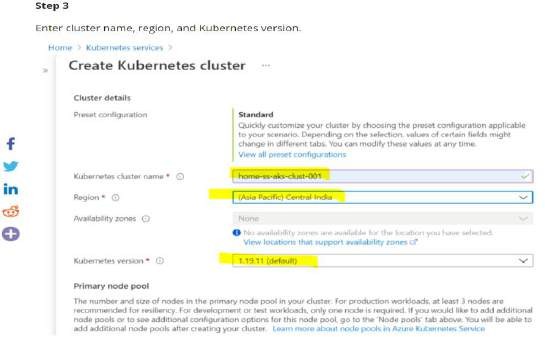
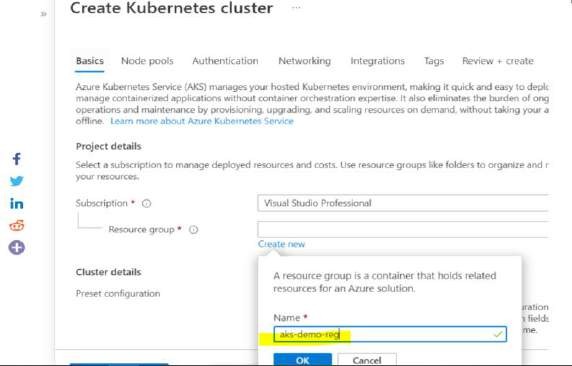
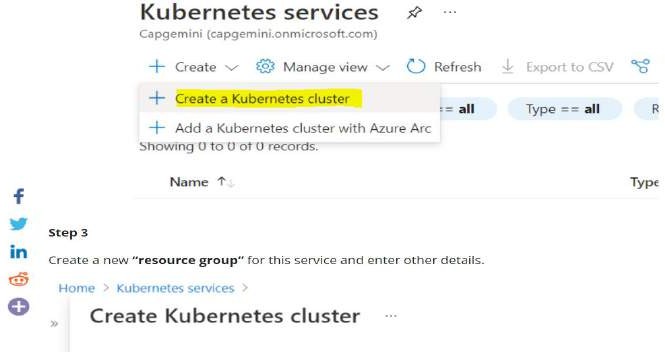
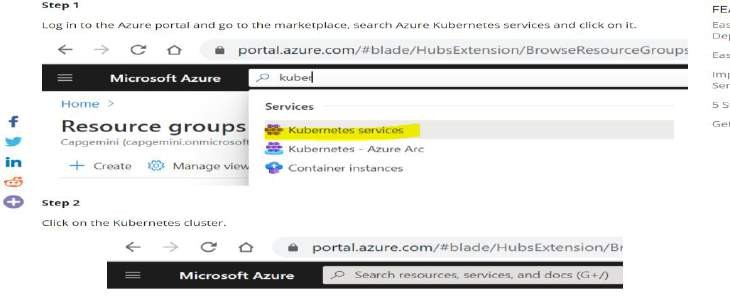
Since version 4.5, NLog has supported Structured Logging beside their event properties. So depending on the target logger and if that target supports structured logging, you can NLog for that purpose.

For Pellerex, we have adopted SeriLog, which from the capability point of view, is at the same level as NLog, but comes with a more set of streamlined set of capabilities when it comes down to Structure Logging [ream more.](https://pellerex.com/blog/structured-logging-with-serilog)

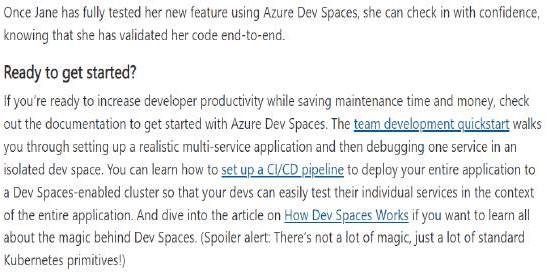
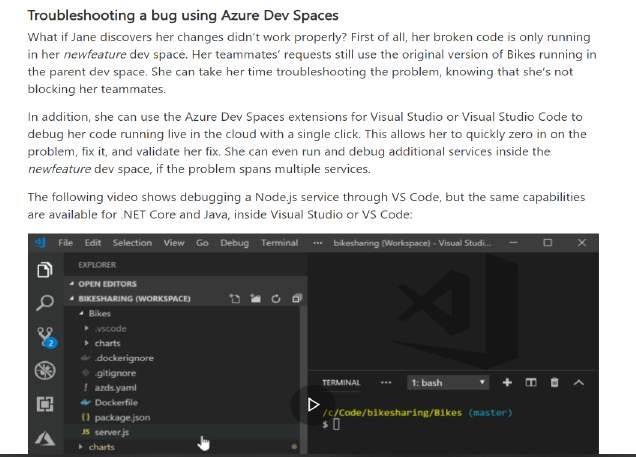
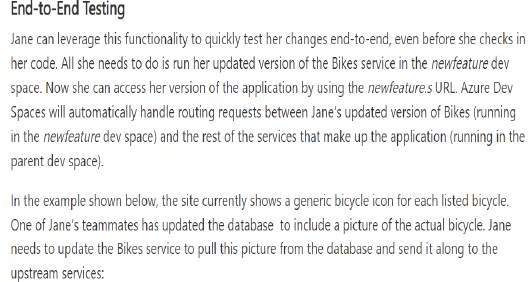
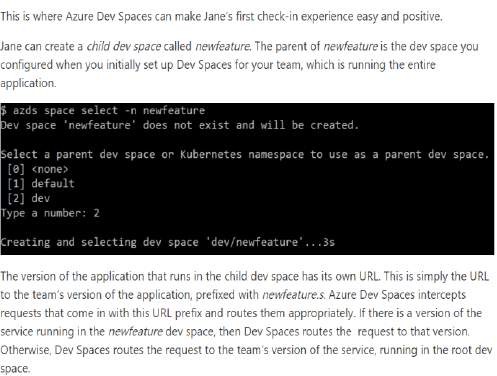
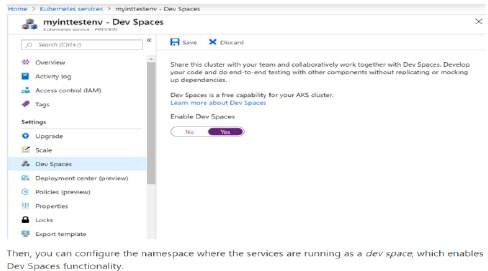
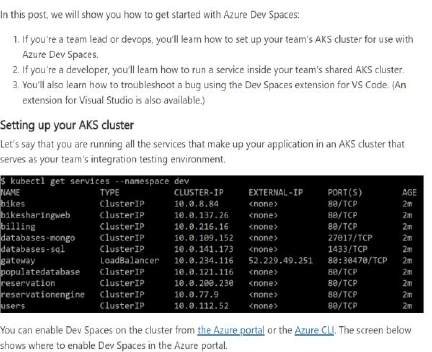
1. **Exceptions:** In Application Insights, it's important to have log records to be classified properly, information as information, and exception as exception. This is important for troubleshooting, monitoring and analytical purposes. Hence we specifically set the exception of the log if the exception is not empty: **logEventInfo.Exception = exception;**



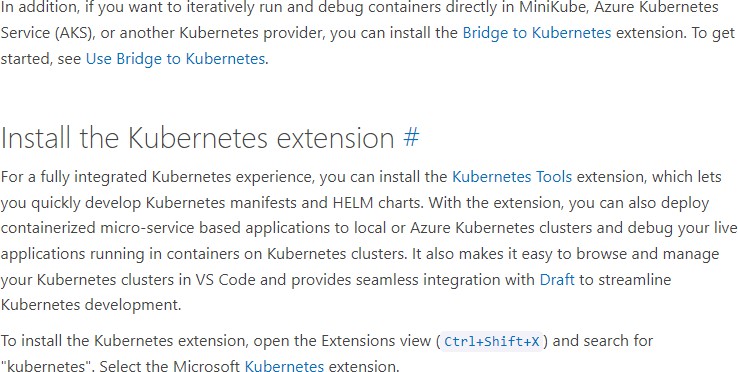
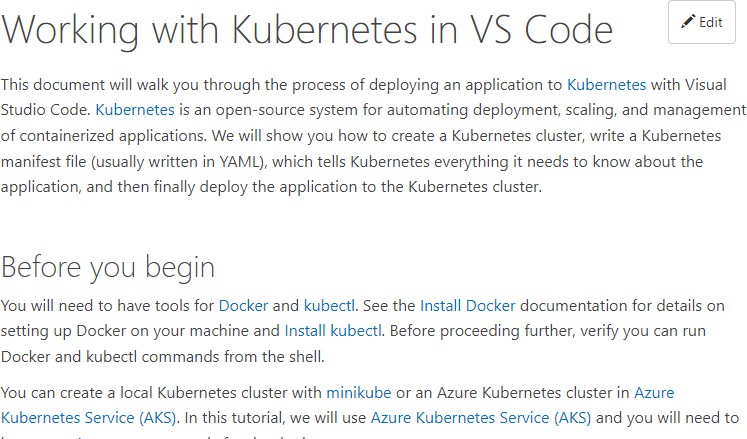
1. a. Create an Azure Kubernetes Service Cluster

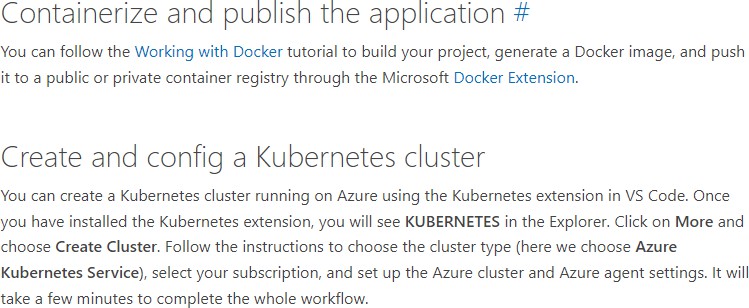


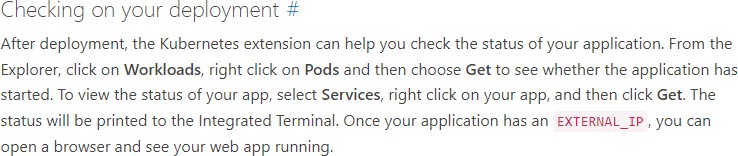
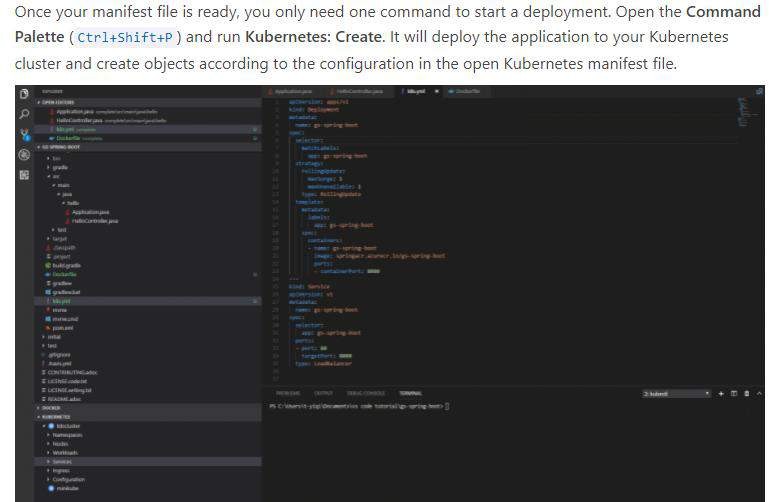
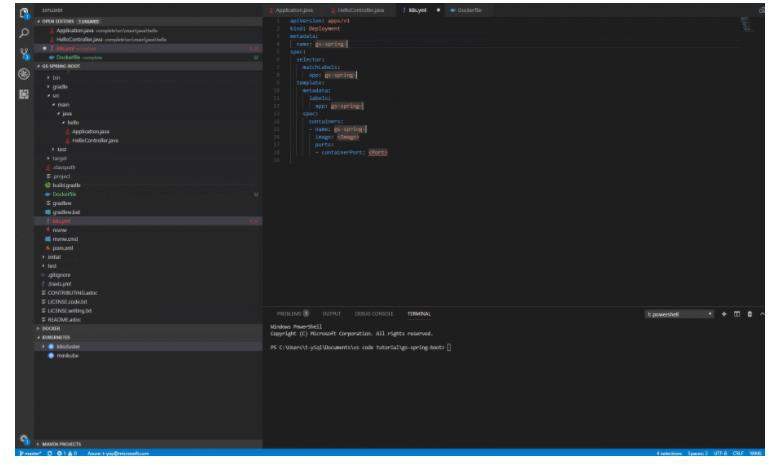
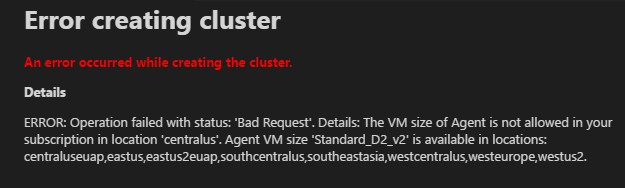
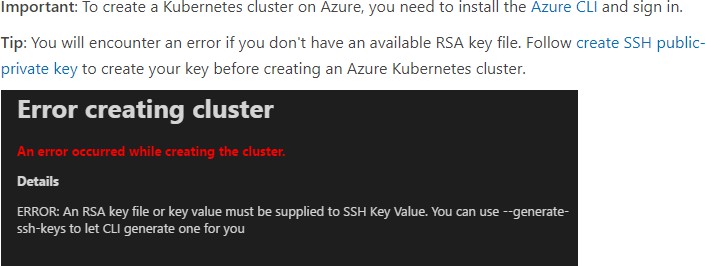
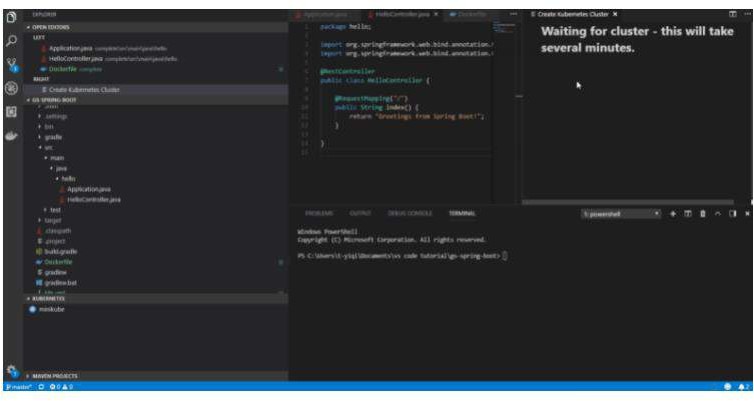
1. Enable Azure Dev Spaces on an AKS Cluster

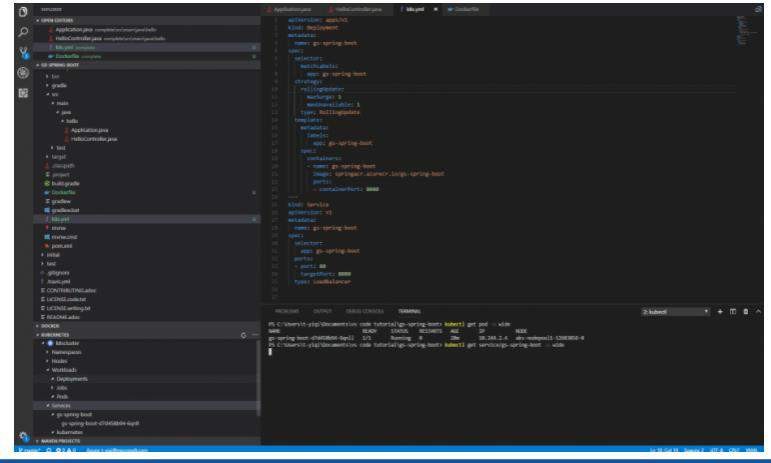


1. Configure Visual Studio to Work with an Azure Kubernetes Service Cluster

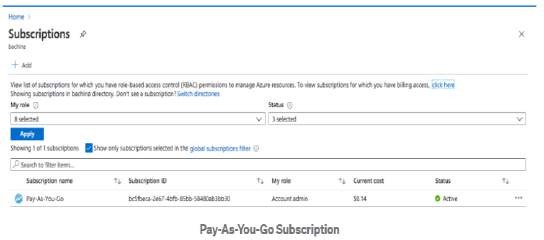
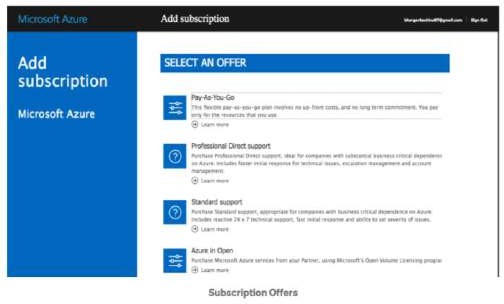








1. Configure Visual Studio Code to Work with an Azure Kubernetes Service Cluster
2. Deploy Application on AKS
   1. Core Web API
   2. Node.js API



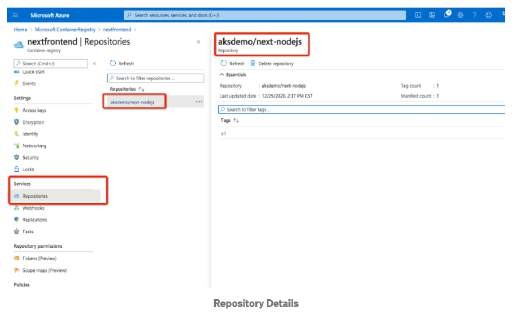
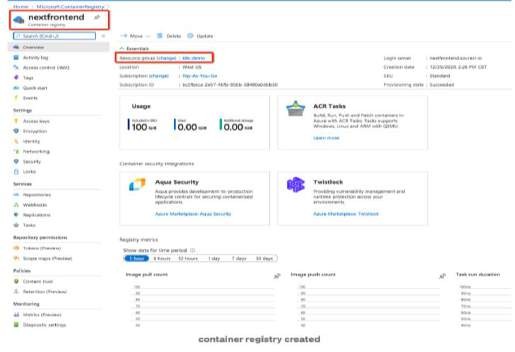
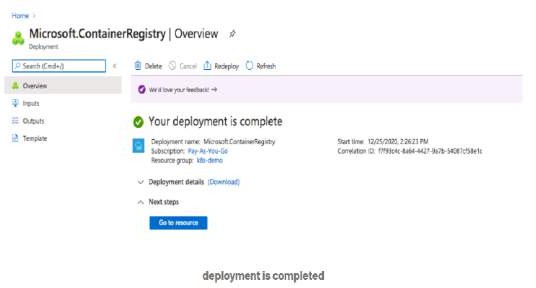
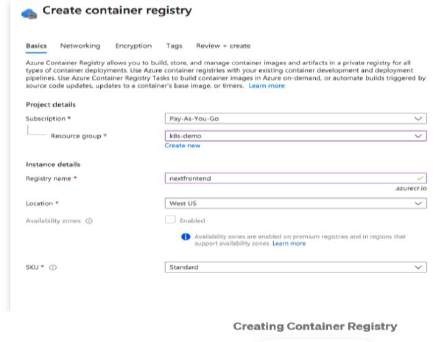
Install Azure CLI and Configure



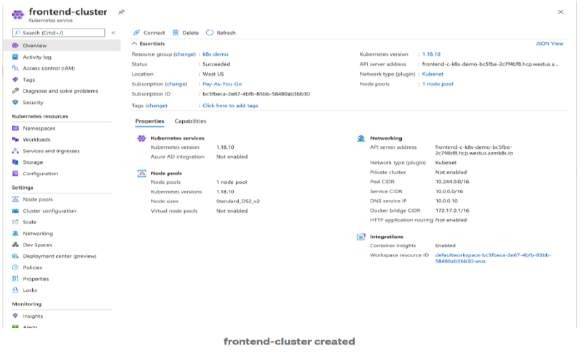
Dockerize the Project



Pushing Docker Image To Container Registry



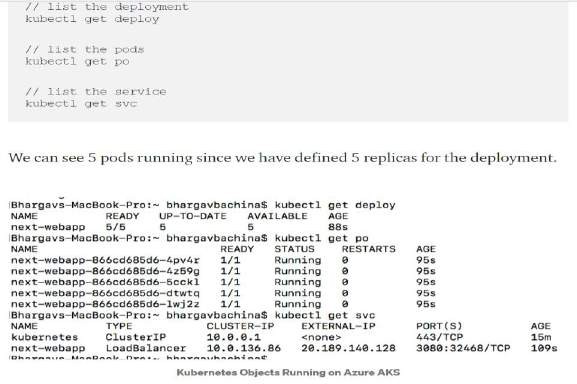
Creating AKS Cluster



Configure Kuebctl With AKS Cluster



Deploy Kubernetes Objects on Azure AKS Cluster



Access the WebApp from the browser

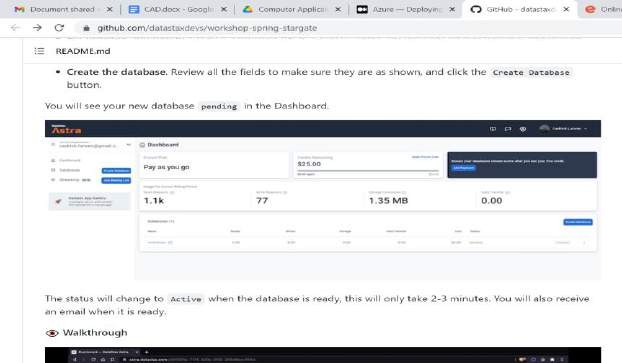


5 Create an AKS cluster

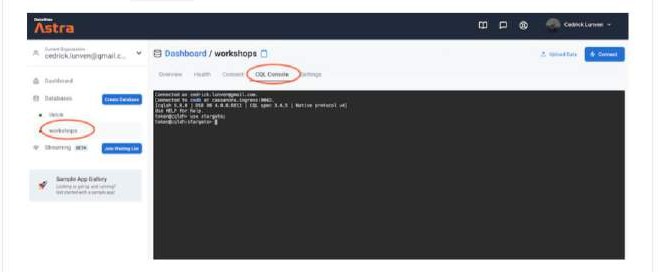
1. from the portal
2. with Azure CLI

Step 1a. If you do have an account yet register and sign In to Astra DB this is FREE and NO CREDIT CARD

Step 1b. Create a "pay as you go" plan



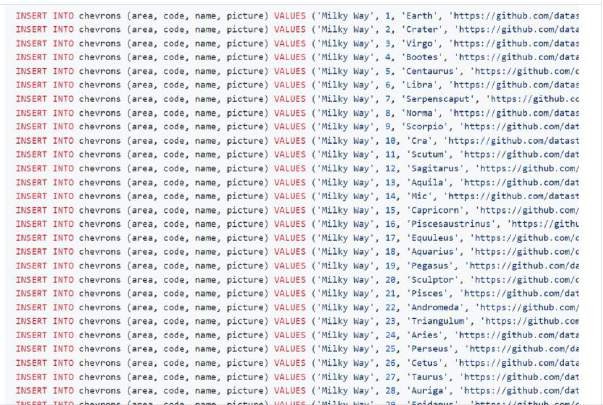
Step2a: Locate and open CQLConsole Step 2b: Navigate to your keyspace



Step 2c: Create Entities



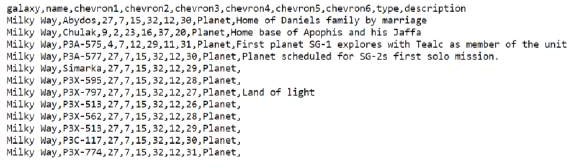
Step 2d: Populate entries



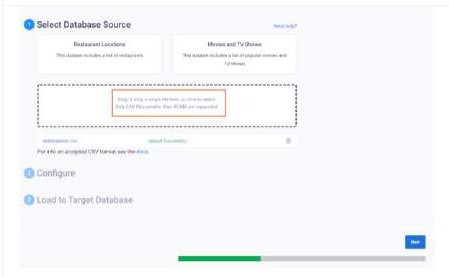
Step 2e: Show the results



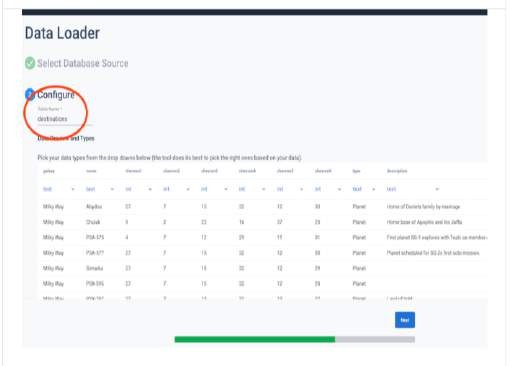
Step 3a: Download the dataset



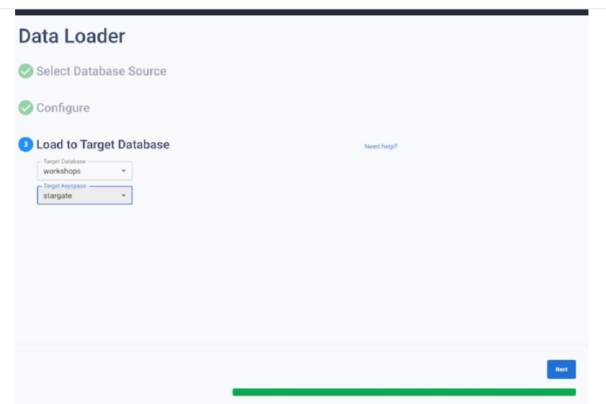
Step 3b: Open Astra Data Loader Step 3c: Upload the dataset



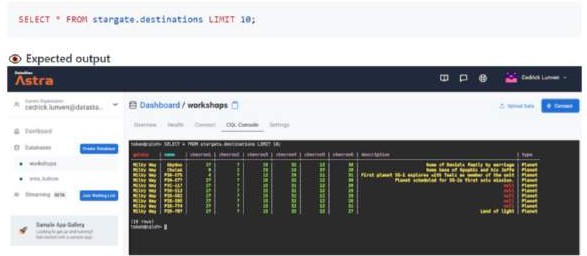
Step 3d: Define the target table



Step 3e: Define the target keyspace



Step 3f: Show Data



1. Create an Application Gateway Using Ocelot and Securing APIs with Azure AD.

# Introducing Ocelot

In this article, we are going to use [Ocelot API Gateway.](https://github.com/ThreeMammals/Ocelot) It is a lightweight, open-source, scalable, and fast API Gateway based on .NET Core and specially designed for microservices architecture. Basically, it is a set of middleware designed to work with ASP.NET Core. It has several features such as routing, caching, security, rate limiting, etc.

# The Order Processing Microservices-Based Application

Let's now put the concepts we've learned thus far into practice by implementing a concrete example. We'll build an order processing application that illustrates how an API Gateway can be used to invoke each service to retrieve customer and product data using the Customer and Product microservice, respectively.Typically, an order processing microservices-based application comprises microservices such as Product, Customer, Order, OrderDetails, etc. In this example, we'll consider a minimalistic microservices-based application. This application will contain an API Gateway and two microservices - the Product and Customer microservice. The application would be simple so that we can focus more on building the API Gateway.

**Prerequisites**To execute the code examples shown in this article, here are the minimum requirements you should have installed in your system:

* + .NET 5 SDK
  + Visual Studio 2019

The solution structure

The application you are going to build will comprise the following projects as part of a single Visual Studio solution:

* + OrderProcessing project - This project represents the API Gateway and is responsible for getting requests from the clients and invoking the microservices.
  + OrderProcessing.Customer project - This project defines the classes and interfaces used to represent the customer microservice.
  + OrderProcessing.Product project - This project defines the types used to represent the product microservice.

The Customer microservice project will comprise the following classes and interfaces:

* + Customer class – This represents the customer entity class.
  + ICustomerRepository interface – This represents the interface for the customer repository.
  + CustomerRepositoryclass – This represents the customer repository class that implements the ICustomerRepository interface.
  + CustomerController class – This class represents the API controller for the Customer microservice.

The Product microservice project will contain the following types:

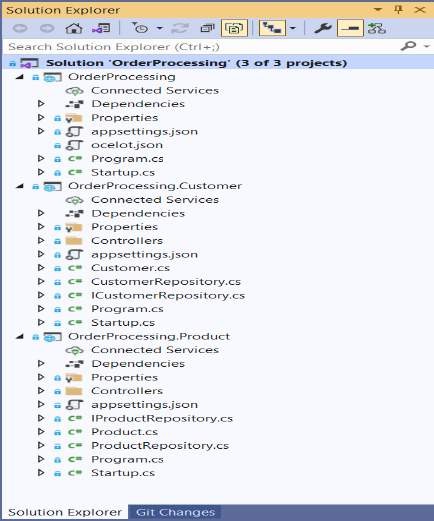
* + Product class – This class represents the product entity.
  + IProductRepository interface – This represents the interface for the product repository.
  + class – This is the product repository class that implements the interface.

IProductRepository

ProductRepository

* + ProductController class – This represents the API controller class for the Product microservice.

The following picture shows how the solution structure of the completed application will look like:



Create the projects for the Order Processing application

Open a command shell and enter the following commands to create the three ASP.NET projects we need:

dotnet new web --framework "net5.0" -o OrderProcessing

dotnet new webapi --framework "net5.0" -o OrderProcessing.Customer dotnet new webapi --framework "net5.0" -o OrderProcessing.Product

While the OrderProcessing project is an empty ASP.NET project, the other two projects are WebAPI projects. Ensure that you delete the default controller and entity classes from these two projects as we don’t need them.

Create the Customer microservice

Create a new file named Customer.cs at the root of the OrderProcessing.Customer project with the following code in there:

// OrderProcessing.Customer/Customer.cs

using System;

namespace OrderProcessing.Customer

{

public class Customer

{

public Guid Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string EmailAddress { get; set; }

}

}

Create the CustomerRepository class

Create an interface named ICustomerRepository in a file named ICustomerRepository.cs at the root of the OrderProcessing.Customer project with the following code in there:

// OrderProcessing.Customer/ICustomerRepository.cs

using System.Collections.Generic; using System.Threading.Tasks;

namespace OrderProcessing.Customer

{

public interface ICustomerRepository

{

public Task<List<Customer>> GetAllCustomers();

}

}

Create the CustomerRepository class that implements the ICustomerRepository interface at the root of the OrderProcessing.Customer project as shown in the following code snippet:

// OrderProcessing.Customer/CustomerRepository.cs

using System;

using System.Collections.Generic; using System.Linq;

using System.Threading.Tasks;

namespace OrderProcessing.Customer

{

public class CustomerRepository : ICustomerRepository

{

private readonly List<Customer> customers = new List<Customer>();

public CustomerRepository()

{

customers.Add(new Customer()

{

Id = Guid.NewGuid(),

FirstName = "Joydip",

LastName = "Kanjilal",

EmailAddress = ["joydipkanjil](mailto:joydipkanjilal@yahoo.com)a[l@yahoo.com"](mailto:joydipkanjilal@yahoo.com)

});

customers.Add(new Customer()

{

Id = Guid.NewGuid(),

FirstName = "Steve",

LastName = "Smith",

EmailAddress = ["st](mailto:stevesmith@yahoo.com)e[vesmith@yahoo.com"](mailto:stevesmith@yahoo.com)

});

}

public Task<List<Customer>> GetAllCustomers()

{

return Task.FromResult(customers);

}

}

}

Create the CustomerController class

In the Controllers folder of the OrderProcessing.Customer project, create an API controller named CustomerController and replace the default code with the following:

// OrderProcessing.Customer/Controllers/CustomerController.cs

using Microsoft.AspNetCore.Mvc; using System.Collections.Generic; using System.Threading.Tasks;

namespace OrderProcessing.Customer.Controllers

{

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

[ApiController]

public class CustomerController : ControllerBase

{

private readonly ICustomerRepository \_customerRepository;

public CustomerController(ICustomerRepository customerRepository)

{

\_customerRepository = customerRepository;

}

[HttpGet]

public async Task<ActionResult<List<Customer>>> GetAllCustomers()

{

return await \_customerRepository.GetAllCustomers();

}

}

}

Create the Product microservice

Create a new file named Product.cs at the root of the OrderProcessing.Product project with the following code in there:

// OrderProcessing.Product/Product.cs

using System;

namespace OrderProcessing.Product

{

public class Product

{

public Guid Id { get; set; }

public string Code { get; set; }

public string Name { get; set; }

public int Quantity\_In\_Stock { get; set; }

public decimal Unit\_Price { get; set; }

}

}

Create the ProductRepository class

Next, you should create a new file called IProductRepository.cs in the OrderProcessing.Product project and write the following code to create the IProductRepository interface.

// OrderProcessing.Product/IProductRepository.cs

using System.Collections.Generic; using System.Threading.Tasks; namespace OrderProcessing.Product

{

public interface IProductRepository

{

public Task<List<Product>> GetAllProducts();

}

}

Create the ProductRepository class that implements the IProductRepository interface at the root of the OrderProcessing.Product project as shown in the following code snippet:

// OrderProcessing.Product/ProductRepository.cs

using System;

using System.Collections.Generic; using System.Threading.Tasks;

namespace OrderProcessing.Product

{

public class ProductRepository : IProductRepository

{

private readonly List<Product> products = new List<Product>();

public ProductRepository()

{

products.Add(new Product

{

Id = Guid.NewGuid(),

Code = "P0001",

Name = "Lenovo Laptop",

Quantity\_In\_Stock = 15,

Unit\_Price = 125000

});

products.Add(new Product

{

Id = Guid.NewGuid(),

Code = "P0002",

Name = "DELL Laptop",

Quantity\_In\_Stock = 25,

Unit\_Price = 135000

});

products.Add(new Product

{

Id = Guid.NewGuid(),

Code = "P0003",

Name = "HP Laptop",

Quantity\_In\_Stock = 20,

Unit\_Price = 115000

});

}

public Task<List<Product>> GetAllProducts()

{

return Task.FromResult(products);

}

}

}

Create the ProductController class

In the Controllers folder of the OrderProcessing.Product project, create an API controller named ProductController and replace the default code with the following:

// OrderProcessing.Product/ProductController.cs

using Microsoft.AspNetCore.Mvc; using System.Collections.Generic; using System.Threading.Tasks;

namespace OrderProcessing.Product.Controllers

{

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

[ApiController]

public class ProductController : ControllerBase

{

private readonly IProductRepository \_productRepository;

public ProductController(IProductRepository customerRepository)

{

\_productRepository = customerRepository;

}

[HttpGet]

public async Task<ActionResult<List<Product>>> GetAllCustomers()

{

return await \_productRepository.GetAllProducts();

}

}

}

# Implement the API Gateway Using Ocelot

Now that the projects have been created with the necessary files in them, let’s implement the API Gateway using Ocelot.Before going any further, you should be aware of the terms upstream and downstream. While upstream refers to the request sent by the client to the API Gateway, downstream is related to the request that the API Gateway sends to a particular microservice.

Install the required package

To work with Ocelot, you must install it in your ASP.NET Core project. In our case, you will install Ocelot in the OrderProcessing project. You can do it by using the NuGet Package Manager inside Visual Studio IDE. Alternatively, you can execute the following command at the Package Manager Console window:

Install-Package Ocelot

Implement routing

An Ocelot API Gateway accepts an incoming HTTP request and forwards it to a downstream service. Ocelot makes use of routes to define how a request is routed from one place to another. Add a new file named ocelot.json to this project with the following content in there:

// OrderProcessing/Ocelot.json

{

"Routes":[

//Customer API{

"DownstreamPathTemplate":"/api/Customer",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"20057"

}

],

"UpstreamPathTemplate":"/Customer",

"UpstreamHttpMethod":[

"GET"

]

},

//Product API{

"DownstreamPathTemplate":"/api/Product",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"32345"

}

],

"UpstreamPathTemplate":"/Product",

"UpstreamHttpMethod":[

"GET"

]

}

]

}

The above configuration specifies the downstream and upstream metadata (scheme, path, ports) for the customer and product microservices. So, while use the upstream metadata to call the endpoints specified here, the request is routed to the appropriate downstream service as specified in the downstream metadata. In other words, the downstream metadata is used to specify the internal service URL to redirect a request to when the API Gateway receives a new request.You should add Ocelot to the service container by calling the AddOcelot method in the ConfigureServices method of the Startup class as shown below:

// OrderProcessing/Startup.cs

// ... existing code

public void ConfigureServices(IServiceCollection services)

{

services.AddOcelot(Configuration);

}

// ... existing code

Next, you should enable Ocelot in the Configure method of the Startup class by calling theUseOcelot extension method as shown here:

// OrderProcessing/Startup.cs

// ... existing code

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseOcelot();

app.UseEndpoints(endpoints => {

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

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

});

});

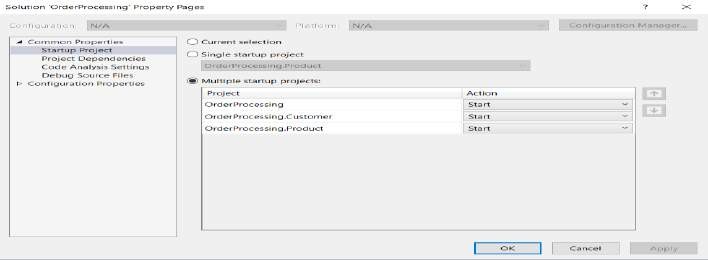
}

// ... existing code

Run the projects

Now make sure that you've made all three projects as startup projects. To do this, follow these steps:

1. In the Solution Explorer window, right-click on the solution file.
2. Click "Properties".
3. In the "Property Pages" window, select the "Multiple startup projects" radio button:

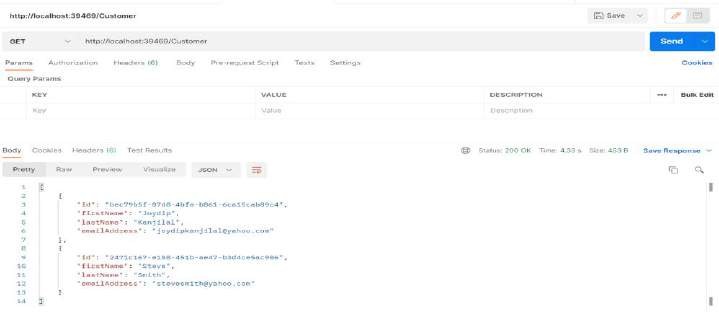


1. Click Ok.

Press the F5 key to run the application. Now send an HTTP Get request to the following URL from [Postman](https://www.postman.com/) or any other HTTP client of your choice:

http://localhost:39469/Customer

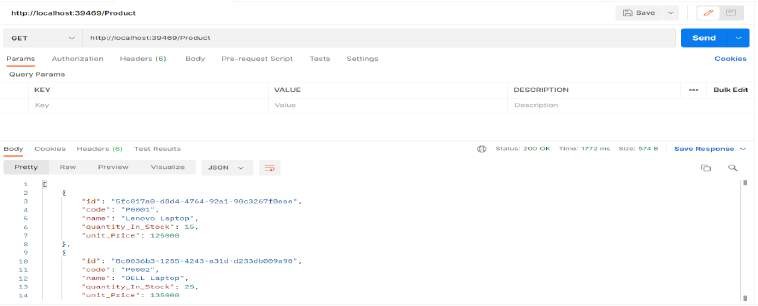
The HTTP Get method of the Customer controller will be executed and the output will look like this:



Send an HTTP Get request from Postman to the following URL:

http://localhost:39469/Product

The request first goes to the API Gateway. Next, the API Gateway routes the request to the correct downstream microservice as specified in ocelot.json. The HTTP Get method named GetAllProducts of the ProductController will be called, and the output will look like this:



Implement rate limiting

Rate limiting is a technique for controlling network traffic. It sets a limit on how many times you can perform a specific activity within a given period - for example, accessing a particular resource, logging into an account, etc. Typically, rate-limiting keeps track of the IP addresses and the time elapsed between requests. The IP address helps determine the source of aparticular request.A rate-limiting solution is adept at tracking the time elapsed between each request and the total number of requests in a particular period. If a single IP address makes excessive requests within a specific timeframe, the rate-limiting solution will reject the requests for a specified period.

In order to prevent your downstream services from being overburdened, Ocelot enables rate- limiting of upstream requests. The following configuration illustrates how you specify rate- limiting in Ocelot:

// OrderProcessing/Ocelot.json

"RateLimitOptions":{

"ClientWhitelist":[

],

"EnableRateLimiting":true,

"Period":"5s",

"PeriodTimespan":1,

"Limit":1,

"HttpStatusCode":429

}

Let us now examine each of these options briefly:

* ClientWhitelist setting - This is an array used to specify the clients that should not be affected by the rate-limiting.
* EnableRateLimiting setting - This is a boolean value, true if you want to enable rate- limiting, false otherwise.
* HttpStatusCode setting - This is used to specify [the HTTP status code that is returned](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429) [when rate limiting occurs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429).
* Period setting - This specifies the duration that the rate limit is applicable, which in turn implies that if you make more requests within this duration than what is allowed, you'll need to wait for the duration specified in the PeriodTimespan.
* PeriodTimespan setting - This is used to specify the duration after which you can retry to connect to a service.
* Limit setting - This specifies the maximum number of requests that are allowed within the duration specified in Period.

Let us assume that rate limiting is applied to the Product microservice only. The updated ocelot.json file will now look like this:

// OrderProcessing/Ocelot.json

{

"Routes":[

//Customer API

{

"DownstreamPathTemplate":"/api/Customer",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"20057"

}

],

"UpstreamPathTemplate":"/Customer",

"UpstreamHttpMethod":[

"GET"

]

},

//Product API

{

"DownstreamPathTemplate":"/api/Product",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"32345"

}

],

"RateLimitOptions":{

"ClientWhitelist":[

],

"EnableRateLimiting":true,

"Period":"5s",

"PeriodTimespan":1,

"Limit":1

},

"UpstreamPathTemplate":"/Product",

"UpstreamHttpMethod":[

"GET"

]

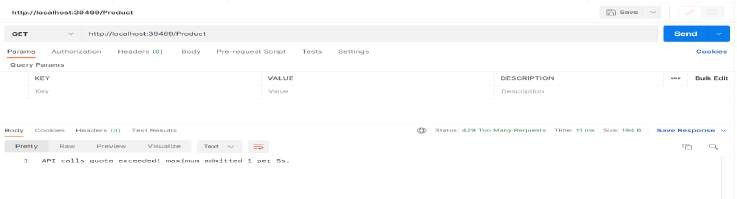
}

]

}

Now, run the application and send frequent requests (more than 1 per 5sec) and you’ll see the

following error:



Implement caching

Caching is a widely popular technique used in web applications to keep data in memory so that the same data may be quickly accessed when required by the application. Ocelot provides support for basic caching. To take advantage of it, you should install the Ocelot.Cache.CacheManager NuGet package as shown below:

Install-Package Ocelot.Cache.CacheManager

Next, you should configure caching using the following code in the ConfigureServices method:

// OrderProcessing/Startup.cs

// ... existing code ...

public void ConfigureServices(IServiceCollection services)

{

services.AddOcelot(Configuration)

.AddCacheManager(x =>

{

x.WithDictionaryHandle();

});

}

// ... existing code ...

Lastly, you should specify caching on a particular route in the route configuration using the following settings:

// OrderProcessing/Ocelot.json

// ... existing settings ...

"Routes":[

//Customer API{

"DownstreamPathTemplate":"/api/Customer",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"20057"

}

],

"FileCacheOptions":{

"TtlSeconds":30,

"Region":"customercaching"

},

"UpstreamPathTemplate":"/Customer",

"UpstreamHttpMethod":[

"GET"

]

}

]

// ... existing settings ...

Here, we've set TtlSeconds to 30 seconds which implies that the cache will expire after this time has elapsed. Note that you should specify your cache configuration in the FileCacheOptions section. The Region setting identifies the area within the cache that will contain the data. This way you can clear that area by using the Ocelot's administration API.

To test this, you can set a breakpoint on the HTTP Get method named GetAllCustomers in the CustomerController class. When you execute the application and send an HTTP Get request to the endpoint, the breakpoint will be hit as usual. However, all subsequent calls to the same endpoint within 30 seconds (this is the duration we've specified) will fetch data, but the breakpoint will not be hit anymore.

Implement correlation ID

Ocelot enables a client to send a request Id in the header to the server. Once this request Id is available in the middleware pipeline, you can log it along with other information. Ocelot can also forward this request Id to the downstream services if required. A correlation ID is a

unique identifier attached to every request and response and used to track requests and responses in a distributed application. You can use either a request Id or a correlation ID when working with Ocelot to track requests.

The primary difference between a request Id and a correlation ID is that while the former uniquely identifies every HTTP request, the latter is a unique identifier attached to a particular request-response chain. While you can use Request-Id for every HTTP request, you can use X-Correlation-Id for an event chain of requests and responses. X-Correlation-Id is the name of the HTTP header attached to the downstream requests used to track HTTP requests that flow through multiple back-end services.

Ocelot must know the URL that it is running on in order to perform certain administration configurations. This is the BaseUrl specified in the ocelot.json file. Note that this URL should be the URL that your clients will see the API Gateway running on.Here's the complete source code of the ocelot.json file for your reference:

// OrderProcessing/Ocelot.json

{

"Routes":[

//Customer API

{

"DownstreamPathTemplate":"/api/Customer",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"20057"

}

],

"FileCacheOptions":{

"TtlSeconds":30,

"Region":"customercaching"

},

"UpstreamPathTemplate":"/Customer",

"UpstreamHttpMethod":[

"GET"

]

},

//Product API

{

"DownstreamPathTemplate":"/api/Product",

"DownstreamScheme":"http",

"DownstreamHostAndPorts":[

{

"Host":"localhost",

"Port":"32345"

}

],

"RateLimitOptions":{

"ClientWhitelist":[

],

"EnableRateLimiting":true,

"Period":"5s",

"PeriodTimespan":1,

"Limit":1

},

"UpstreamPathTemplate":"/Product",

"UpstreamHttpMethod":[

"GET"

]

}

],

"GlobalConfiguration":{

"RequestIdKey":"X-Correlation-Id",

"BaseUrl":"http://localhost:39469"

}

}

# Conclusion

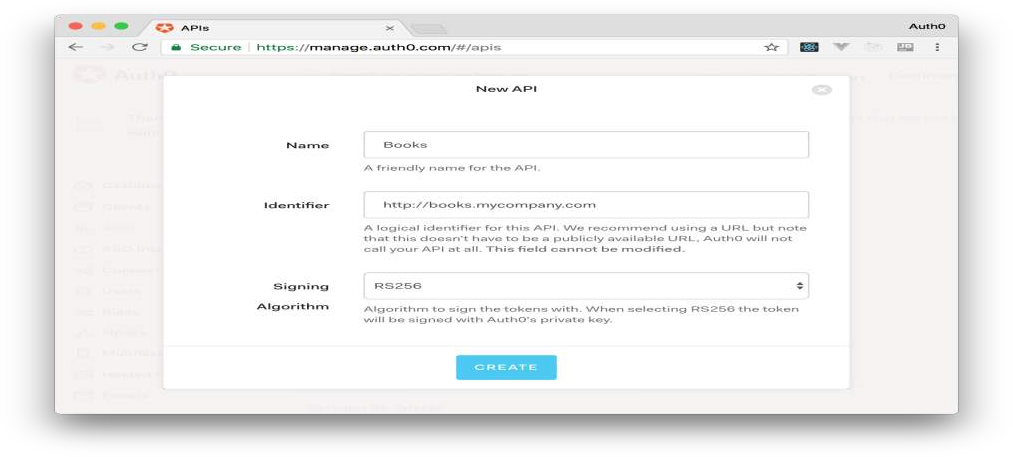
Choosing an exemplary architecture for the needs of your business is the first and foremost step in building applications that are flexible, scalable, and high performant. One of the most significant advantages of microservices architecture is its support for heterogeneous platforms and technologies.

Your API Gateway can manage concerns such as security, rate limiting, performance, and scalability. However, you should be aware of handling the complexity it brings in and the risk of a single point of failure. Besides, there is a learning curve when you're building microservices-based applications using an API Gateway. Possible performance degradation is yet another concern that you must handle.The complete source code of the *OrderProcessing* application built throughout this article is available [here.](https://github.com/auth0-blog/api-gateway-aspnetcore-ocelot)

# Aside: Securing ASP.NET Core with Auth0

Securing ASP.NET Core applications with Auth0 is easy and brings a lot of great features to the table. With [Auth0,](https://auth0.com/) you only have to write a few lines of code to get a solid [identity](https://auth0.com/user-management) [management solution](https://auth0.com/user-management), [single sign-on](https://auth0.com/docs/sso/single-sign-on), support for [social identity providers (like Facebook,](https://auth0.com/docs/identityproviders) [GitHub, Twitter, etc.),](https://auth0.com/docs/identityproviders) and support for [enterprise identity providers (like Active Directory,](https://auth0.com/enterprise) [LDAP, SAML, custom, etc.).](https://auth0.com/enterprise)

On ASP.NET Core, you need [to create an API in your Auth0 Management Dashboard](https://auth0.com/docs/apis) and change a few things on your code. To create an API, you need to [sign up for a free Auth0](https://auth0.com/signup) [account.](https://auth0.com/signup) After that, you need to go to [the API section of the dashboard](https://manage.auth0.com/%23/apis) and click on "Create API". On the dialog shown, you can set the *Name* of your API as "Books", the *Identifier* as "[http://books.mycompany.com](http://books.mycompany.com/)", and leave the *Signing Algorithm* as "RS256".



After that, you have to add the call to services.AddAuthentication() in the ConfigureServices() method of the Startup class as follows:

string authority = $"https://{Configuration["Auth0:Domain"]}/"; string audience = Configuration["Auth0:Audience"];

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>

{

options.Authority = authority;

options.Audience = audience;

});

In the body of the Configure() method of the Startup class, you also need to add an invocation to app.UseAuthentication() and app.UseAuthorization() as shown below:

app.UseRouting(); app.UseAuthentication(); app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

Make sure you invoke these methods in the order shown above. It is essential so that everything works properly.

Finally, add the following element to the appsettings.json configuration file:

{

"Logging": {

// ...

},

"Auth0": {

"Domain": "YOUR\_DOMAIN",

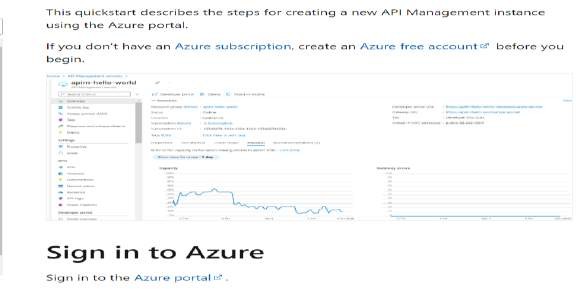
"Audience": "YOUR\_AUDIENCE"

}

}

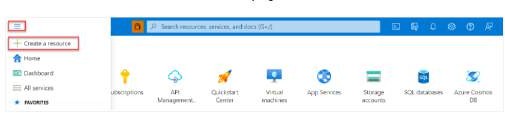
Note: Replace the placeholders YOUR\_DOMAIN and YOUR\_AUDIENCE with the actual values for the domain that you specified when creating your Auth0 account and the *Identifier* you assigned to your API.

1. Create a database design for Microservices an application using the database. 8 a. Create an API management service

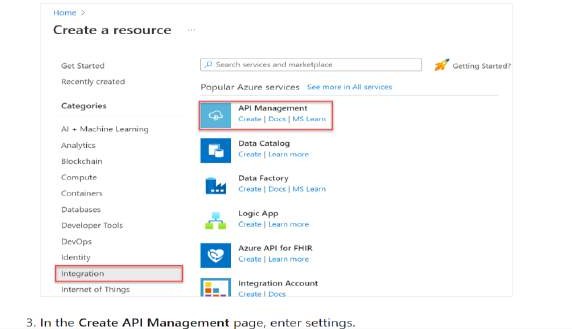


# Create a new service

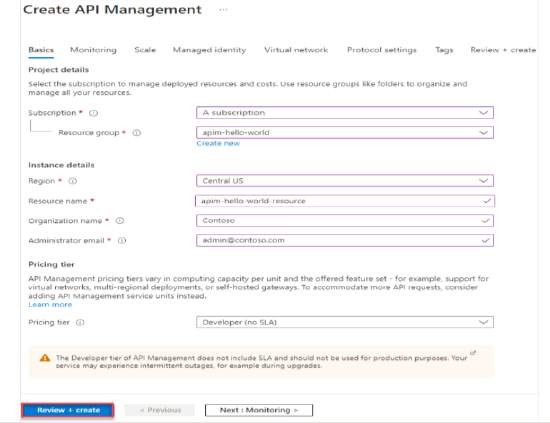
1. From the Azure portal menu, select Create a resource. You can also select Create a resource on the Azure Home page.

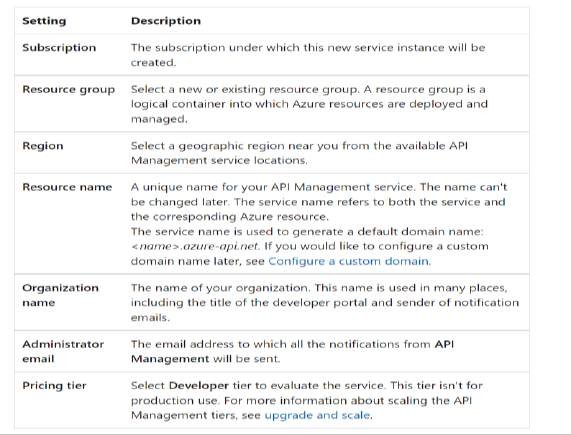


1. On the Create a resource page, select Integration > API Management.



1. In the Create API Management page, enter settings.

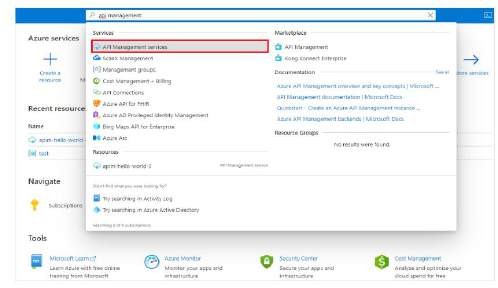




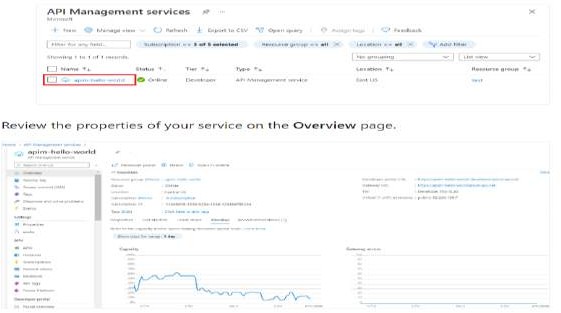
1. Select Review + create.

# Go to your API Management instance

* 1. In the Azure portal, search for and select API Management services.

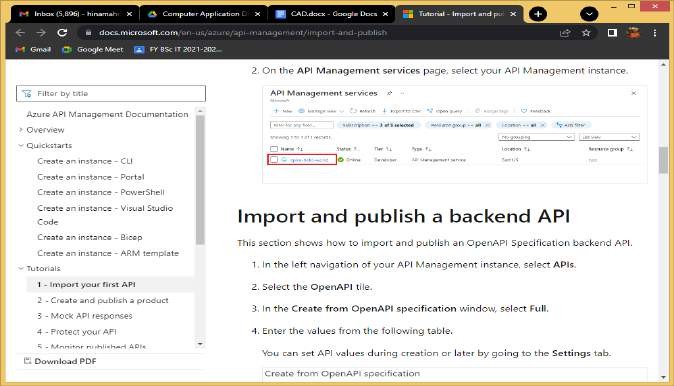


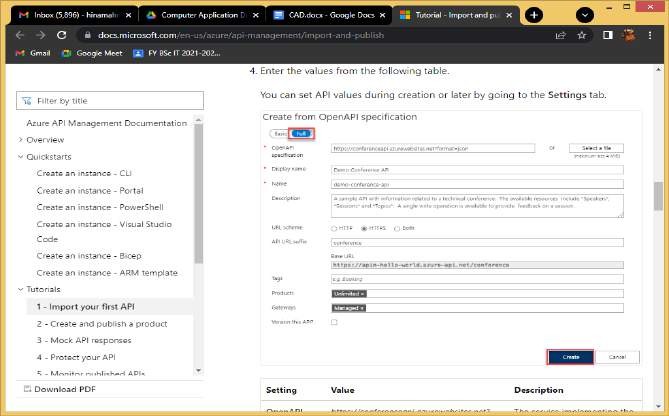
* 1. On the API Management services page, select your API Management instance.

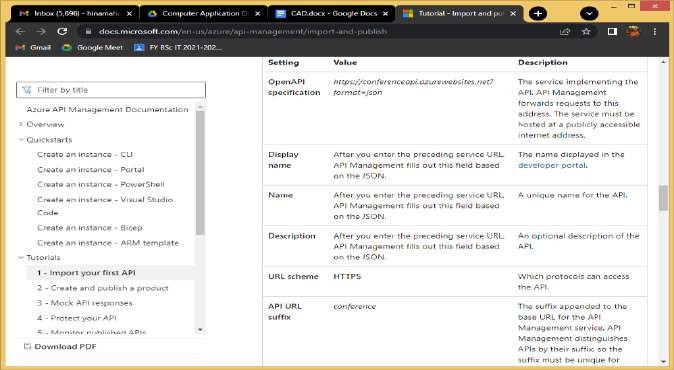


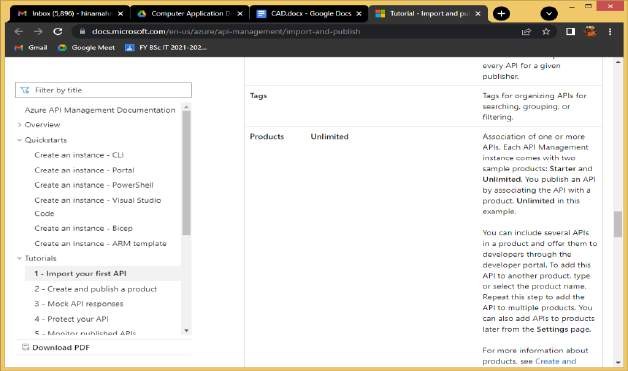
8 b. Create an API gateway service

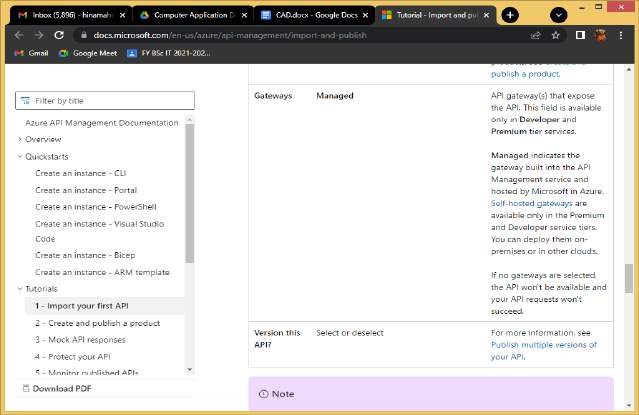


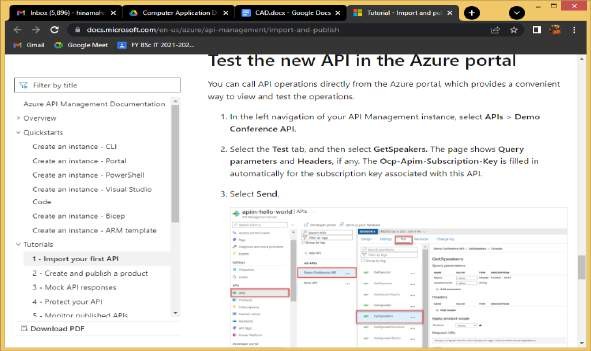


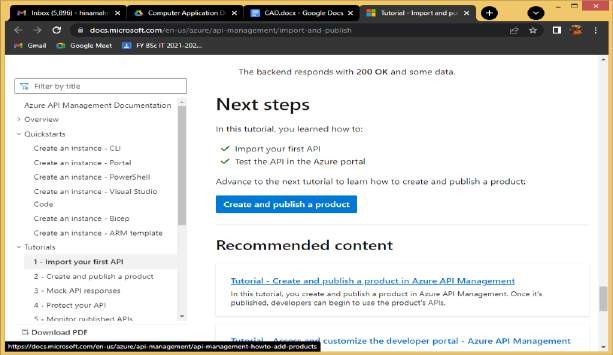




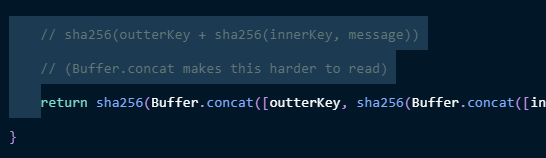








1. Demonstrate
   1. Securing APIs with Azure Active Directory.
      1. In the Azure portal, search for and select App registrations.
      2. Choose your client app. ...
      3. Select Add a Permission.
      4. Under Select an API, select My APIs, and then find and select your backend-app.
      5. Select Delegated Permissions, then select the appropriate permissions to your backend-app.
   2. Issuing a custom JWT token using a symmetric signing key



* 1. Pre-Authentication in Azure API Management
  2. AWS API Gateway Authorizer

{

"version": "2.0",

"type": "REQUEST",

"routeArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/request",

"identitySource": ["user1", "123"],

"routeKey": "$default",

"rawPath": "/my/path",

"rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",

"cookies": ["cookie1", "cookie2"],

"headers": {

"Header1": "value1",

"Header2": "value2"

},

"queryStringParameters": {

"parameter1": "value1,value2",

"parameter2": "value"

},

"requestContext": {

"accountId": "123456789012",

"apiId": "api-id",

"authentication": {

"clientCert": {

"clientCertPem": "CERT\_CONTENT",

"subjectDN": ["www.exampl](http://www.example.com/)e[.com"](http://www.example.com/),

"issuerDN": "Example issuer",

"serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",

"validity": {

"notBefore": "May 28 12:30:02 2019 GMT",

"notAfter": "Aug 5 09:36:04 2021 GMT"

}

}

},

"domainName": "id.execute-api.us-east-1.amazonaws.com",

"domainPrefix": "id",

"http": {

"method": "POST",

"path": "/my/path",

"protocol": "HTTP/1.1",

"sourceIp": "IP",

"userAgent": "agent"

},

"requestId": "id",

"routeKey": "$default",

"stage": "$default",

"time": "12/Mar/2020:19:03:58 +0000",

"timeEpoch": 1583348638390

},

"pathParameters": { "parameter1": "value1" },

"stageVariables": { "stageVariable1": "value1", "stageVariable2": "value2" }

}

# Lambda authorizer response format

The payload format version also determines the structure of the response that you must return from your Lambda function.

# Lambda function response for format 1.0

If you choose the 1.0 format version, Lambda authorizers must return an IAM policy that allows or denies access to your API route. You can use standard IAM policy syntax in the policy. For examples of IAM policies, see [Control access for invoking an API.](https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-control-access-using-iam-policies-to-invoke-api.html) The context object is optional. You can pass context properties to Lambda integrations or access logs by using $context.authorizer.*property*. To learn more, see [Customizing HTTP API access logs](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html).

{

"principalId": "abcdef", *// The principal user identification associated with the token sent by the client.*

"policyDocument": {

"Version": "2012-10-17",

"Statement": [

{

"Action": "execute-api:Invoke",

"Effect": "Allow|Deny",

"Resource": "arn:aws:execute- api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/[{resource}/[{child-resources}]]"

}

]

},

"context": {

"exampleKey": "exampleValue"

}

}

# Lambda function response for format 2.0

If you choose the 2.0 format version, you can return a Boolean value or an IAM policy that uses standard IAM policy syntax from your Lambda function. To return a Boolean value, enable simple responses for the authorizer. The following examples demonstrate the format that you must code your Lambda function to return. The context object is optional. You can pass context properties to Lambda integrations or access logs by using

$context.authorizer.*property*. To learn more, see [Customizing HTTP API access logs](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html).

* Simple response
* IAM policy

{

"isAuthorized": true/false,

"context": {

"exampleKey": "exampleValue"

}

}

# Example Lambda authorizer functions

The following example Node.js Lambda functions demonstrate the required response formats you need to return from your Lambda function for the 2.0 payload format version.

* Simple response
* IAM policy

exports.handler = async(event) => {

let response = {

"isAuthorized": false,

"context": {

"stringKey": "value",

"numberKey": 1,

"booleanKey": true,

"arrayKey": ["value1", "value2"],

"mapKey": {"value1": "value2"}

}

};

if (event.headers.authorization === "secretToken") {

response = {

"isAuthorized": true,

"context": {

"stringKey": "value",

"numberKey": 1,

"booleanKey": true,

"arrayKey": ["value1", "value2"],

"mapKey": {"value1": "value2"}

}

};

}

return response;

};

# Identity sources

You can optionally specify identity sources for a Lambda authorizer. Identity sources specify the location of data that's required to authorize a request. For example, you can specify header or query string values as identity sources. If you specify identity sources, clients must include them in the request. If the client's request doesn't include the identity sources, API Gateway doesn't invoke your Lambda authorizer, and the client receives a 401 error. The following identity sources are supported:

# Selection expressions

|  |  |  |
| --- | --- | --- |
| **T** | **E** | **N** |
| **y** | **x** | **o** |
| **p** | **a** | **t** |
| **e** | **m p l e** | **e s** |
| H | $ | H |
| e | r | e |
| a | e | a |
| d | q | d |
| e | u | e |

|  |  |  |
| --- | --- | --- |
| r v a l u e | e s t  .  h e a d e r  .  *n a m e* | r n a m e s a r e c a s e  -  i n s e n s i t i v e  . |
| Q  u e r y s | $ r e q u e | Q  u e r y s |

|  |  |  |
| --- | --- | --- |
| t r i n g v a l u e | s t  .  q u e r y s t r i n g  .  *n a m e* | t r i n g n a m e s a r e c a s e  -  s e n s i t i v e  . |
| C  o n t | $ c o n | T  h e v |

|  |  |  |
| --- | --- | --- |
| e | t | a |
| x | e | l |
| t | x | u |
| v | t | e |
| a | . | o |
| r | *v* | f |
| i | *a* | a |
| a | *r* | s |
| b | *i* | u |
| l | *a* | p |
| e | *b* | p |
|  | *l* | o |
|  | *e* | r |
|  | *N* | t |
|  | *a* | e |
|  | *m* | d |
|  | *e* | [c](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [o](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [n](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [t](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [e](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [x](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [t](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [v](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [a](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [r](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [i](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [a](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [b](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [l](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | [e](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging-variables.html) |
|  |  | . |

|  |  |  |
| --- | --- | --- |
| S | $ | T |
| t | s | h |
| a | t | e |
| g | a | v |
| e | g | a |
| v | e | l |
| a | V | u |
| r | a | e |
| i | r | o |
| a | i | f |
| b | a | a |
| l | b | [s](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
| e | l | [t](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | e | [a](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | s | [g](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | . | [e](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *v* | [v](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *a* | [a](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *r* | [r](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *i* | [i](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *a* | [a](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *b* | [b](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *l* | [l](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *e* | [e](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-stages.html#http-api-stages.stage-variables) |
|  | *N* | . |
|  | *a* |  |
|  | *m* |  |
|  | *e* |  |

# Caching authorizer responses

You can enable caching for a Lambda authorizer by specifying an [authorizerResultTtlInSeconds](https://docs.aws.amazon.com/apigatewayv2/latest/api-reference/apis-apiid-authorizers.html#apis-apiid-authorizers-prop-createauthorizerinput-authorizerresultttlinseconds). When caching is enabled for an authorizer, API Gateway uses the authorizer's identity sources as the cache key. If a client specifies the same parameters in identity sources within the configured TTL, API Gateway uses the cached authorizer result, rather than invoking your Lambda function.

To enable caching, your authorizer must have at least one identity source.

If you enable simple responses for an authorizer, the authorizer's response fully allows or denies all API requests that match the cached identity source values. For more granular permissions, disable simple responses and return an IAM policy.

By default, API Gateway uses the cached authorizer response for all routes of an API that use the authorizer. To cache responses per route, add $context.routeKey to your authorizer's identity sources.

# Create a Lambda authorizer

When you create a Lambda authorizer, you specify the Lambda function for API Gateway to use. You must grant API Gateway permission to invoke the Lambda function by using either the function's resource policy or an IAM role. For this example, we update the resource policy for the function so that it grants API Gateway permission to invoke our Lambda function.

aws apigatewayv2 create-authorizer \

--api-id *abcdef123* \

--authorizer-type REQUEST \

--identity-source '*$request.header.Authorization*' \

--name lambda-authorizer \

--authorizer-uri 'arn:aws:apigateway:*us-west-2*:lambda:path/2015-03- 31/functions/*arn:aws:lambda:us-west-2:123456789012:function:my-function*/invocations' \

--authorizer-payload-format-version '*2.0*' \

--enable-simple-responses

The following command grants API Gateway permission to invoke your Lambda function. If API Gateway doesn't have permission to invoke your function, clients receive a 500 Internal Server Error.

aws lambda add-permission \

--function-name *my-authorizer-function* \

--statement-id apigateway-invoke-permissions-abc123 \

--action lambda:InvokeFunction \

--principal apigateway.amazonaws.com \

--source-arn "arn:aws:execute-api:us-west-2:123456789012:*api-id/*authorizers/*authorizer- id*"

After you've created an authorizer and granted API Gateway permission to invoke it, update your route to use the authorizer.

aws apigatewayv2 update-route \

--api-id *abcdef123* \

--route-id *acd123* \

--authorization-type CUSTOM \

--authorizer-id *de**f123*

# Troubleshooting Lambda authorizers

If API Gateway can't invoke your Lambda authorizer, or your Lambda authorizer returns a response in an invalid format, clients receive a 500 Internal Server Error.

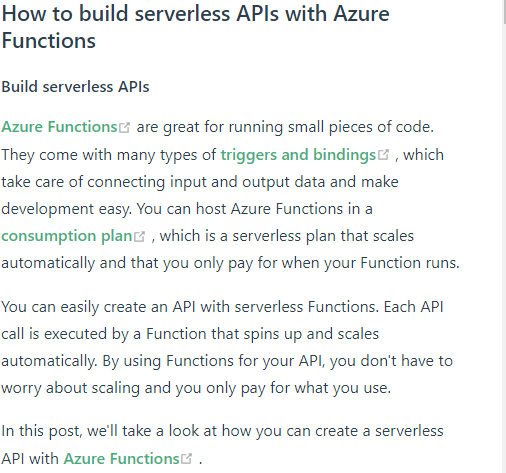
To troubleshoot errors, [enable access logging](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-logging.html) for your API stage. Include the

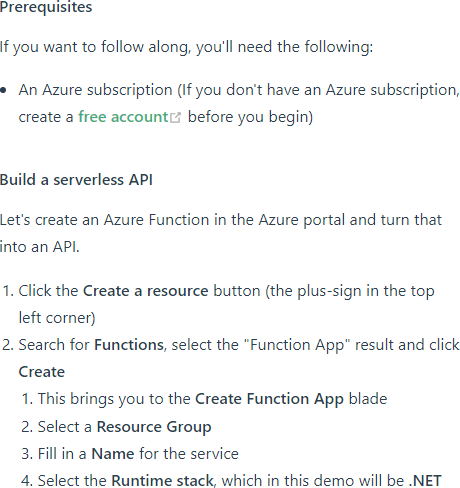
$context.authorizer.error logging variable in your log format.

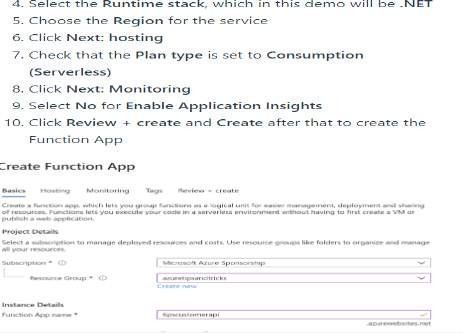
If the logs indicate that API Gateway doesn't have permission to invoke your function, update your function's resource policy or provide an IAM role to grant API Gateway permission to invoke your authorizer.

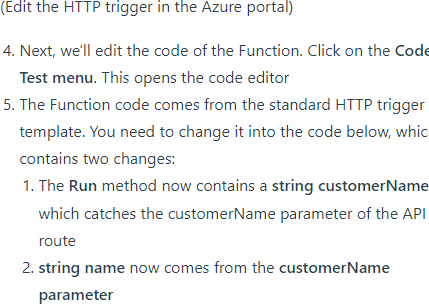
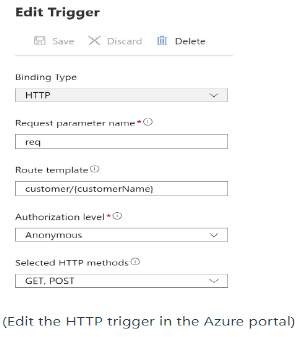
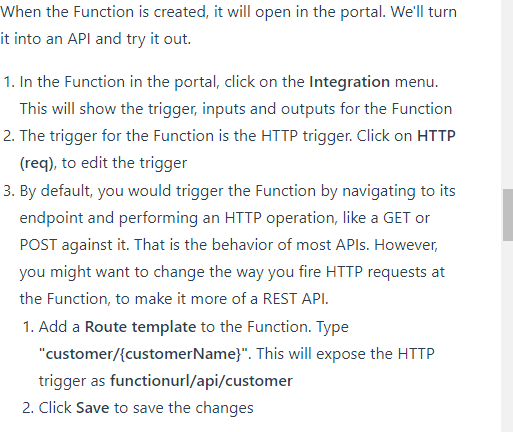
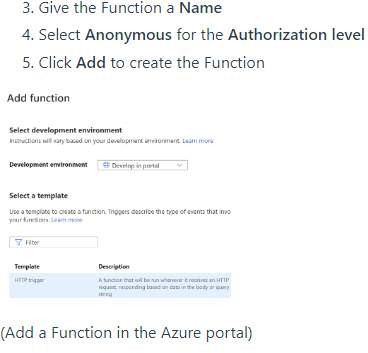
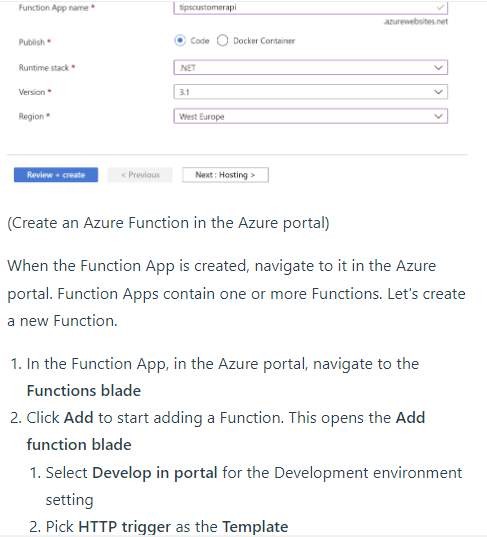
If the logs indicate that your Lambda function returns an invalid response, verify that your Lambda function returns a response in the [required format](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-lambda-authorizer.html#http-api-lambda-authorizer.payload-format-response).

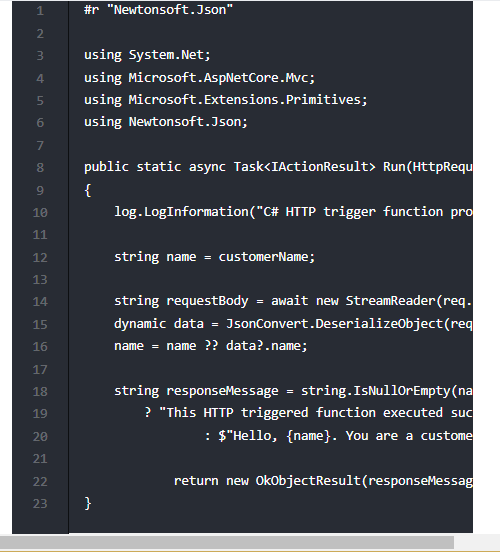
1. Create a serverless API using Azure functions

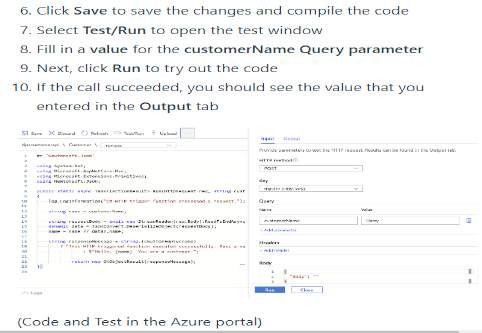


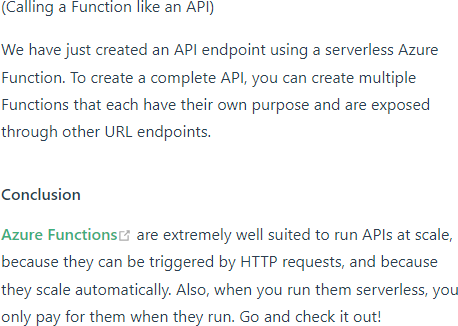
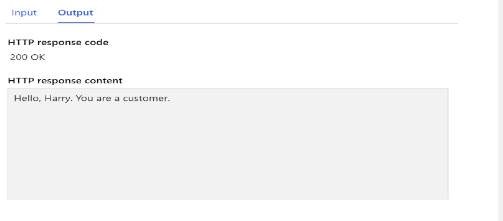
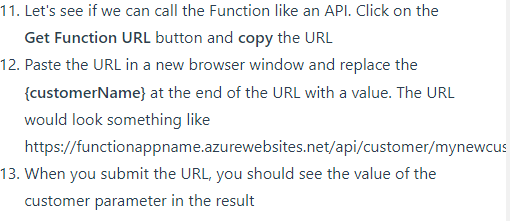




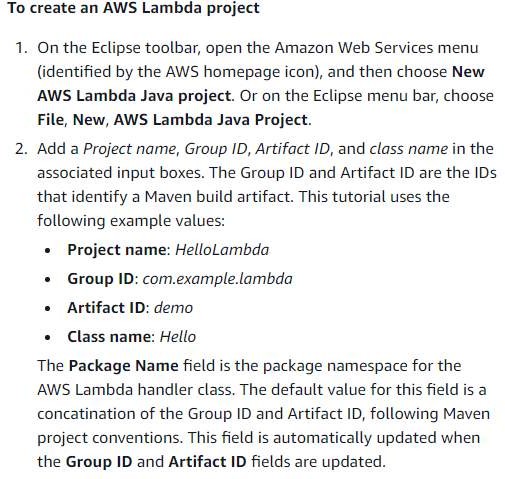


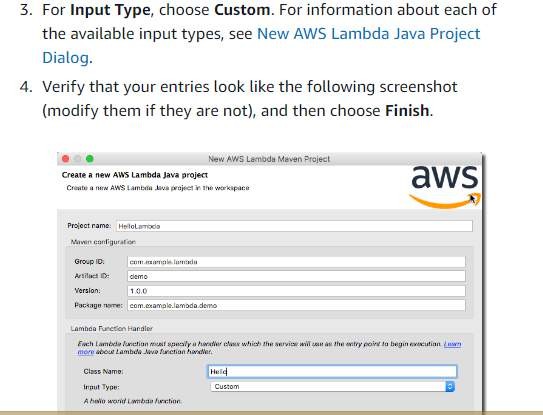


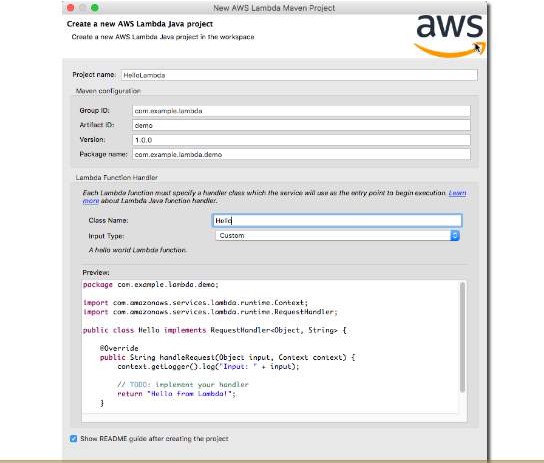


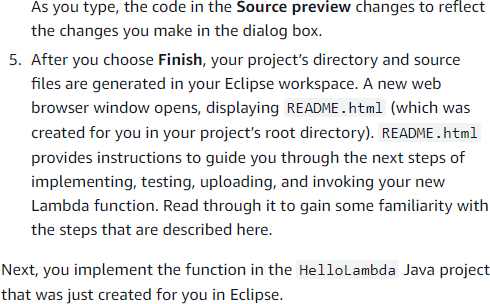


1. Create an AWS Lambda function



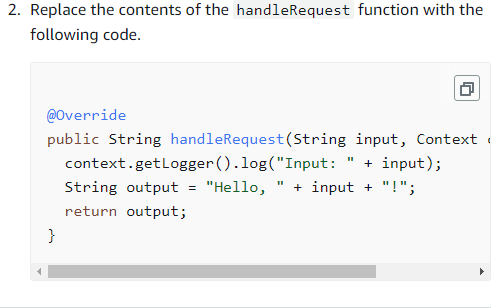




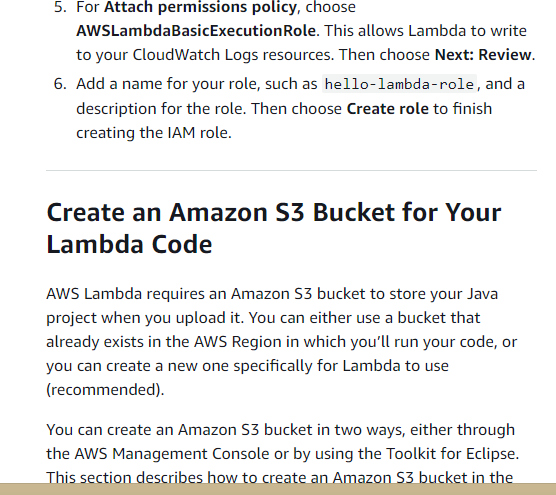


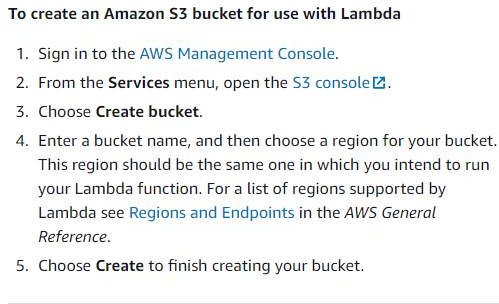


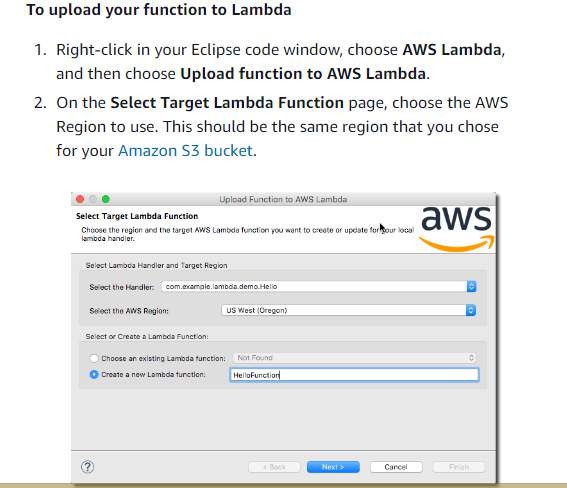


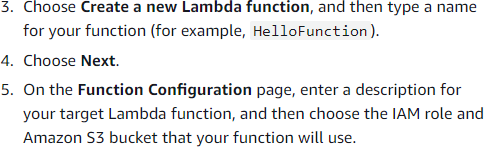


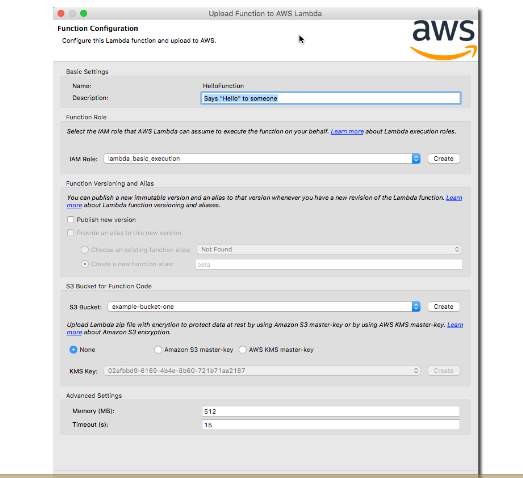


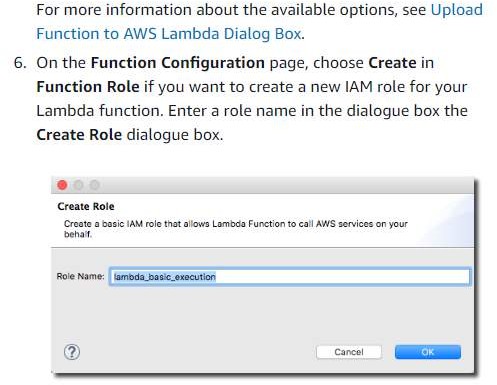


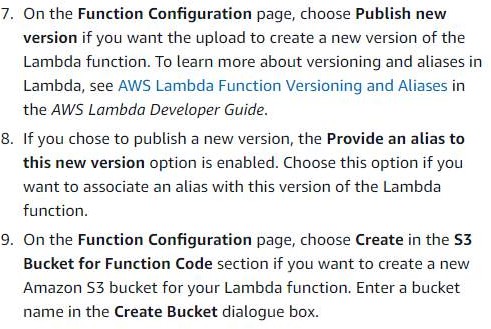


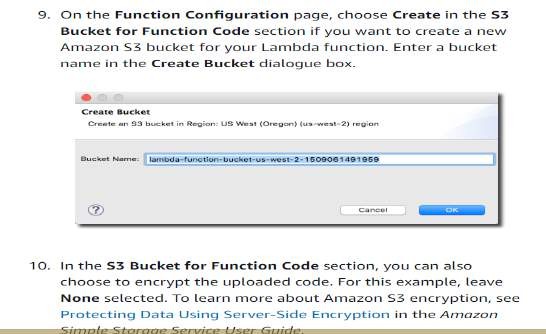


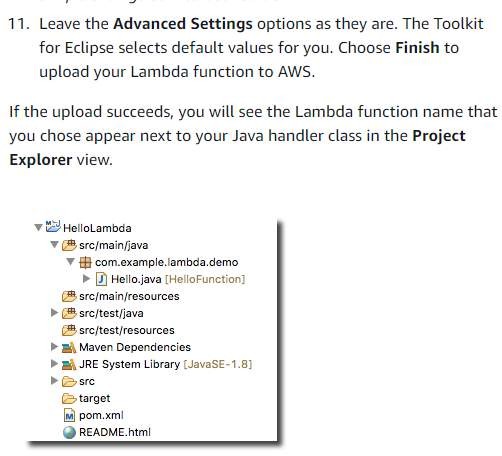


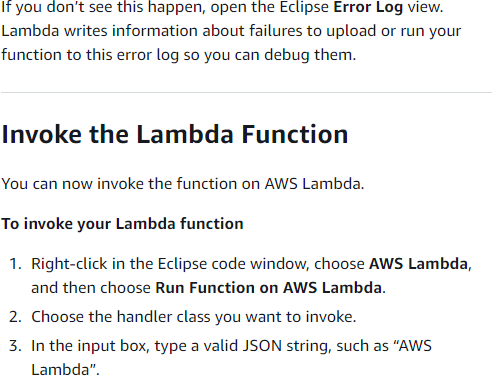


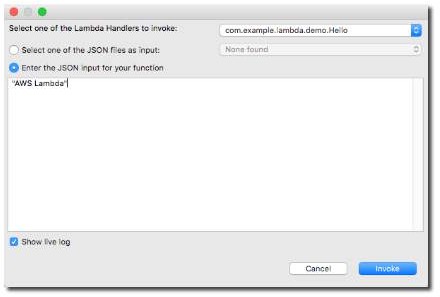


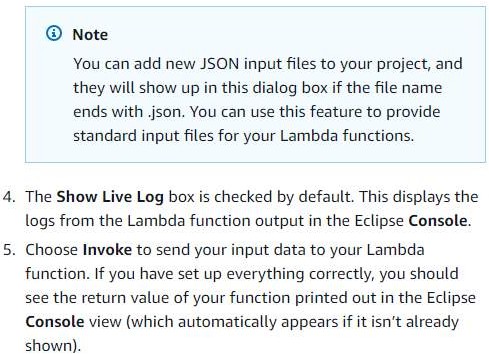




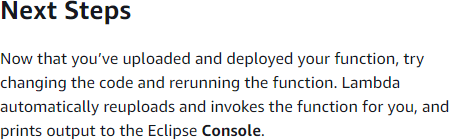












<https://docs.aws.amazon.com/toolkit-for-eclipse/v1/user-guide/lambda-tutorial.html>

1. Build AWS Lambda with AWS API gateway

# Step 1: Create a Lambda function

You use a Lambda function for the backend of your API. Lambda runs your code only when needed and scales automatically, from a few requests per day to thousands per second.

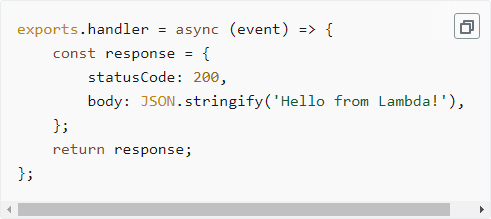
For this example, you use the default Node.js function from the Lambda console. To create a Lambda function

1. Sign in to the Lambda console at [https://console.aws.amazon.com/lambda.](https://console.aws.amazon.com/lambda)
2. Choose Create function.
3. For Function name, enter **my-function**.
4. Choose Create function.

The example function returns a 200 response to clients, and the text Hello from Lambda!.

You can modify your Lambda function, as long as the function's response aligns with the [format that API Gateway requires.](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-develop-integrations-lambda.html#http-api-develop-integrations-lambda.response)

The default Lambda function code should look similar to the following:



# Step 2: Create an HTTP API

Next, you create an HTTP API. API Gateway also supports REST APIs and WebSocket APIs, but an HTTP API is the best choice for this exercise. HTTP APIs have lower latency and lower cost than REST APIs. WebSocket APIs maintain persistent connections with clients for full-duplex communication, which isn't required for this example.

The HTTP API provides an HTTP endpoint for your Lambda function. API Gateway routes requests to your Lambda function, and then returns the function's response to clients.

To create an HTTP API

1. Sign in to the API Gateway console at [https://console.aws.amazon.com/apigateway.](https://console.aws.amazon.com/apigateway)
2. Do one of the following:
   * To create your first API, for HTTP API, choose Build.
   * If you've created an API before, choose Create API, and then choose Build for HTTP API.
3. For Integrations, choose Add integration.
4. Choose Lambda.
5. For Lambda function, enter **my-function**.
6. For API name, enter **my-http-api**.
7. Choose Next.
8. Review the *route* that API Gateway creates for you, and then choose Next.
9. Review the *stage* that API Gateway creates for you, and then choose Next.
10. Choose Create.

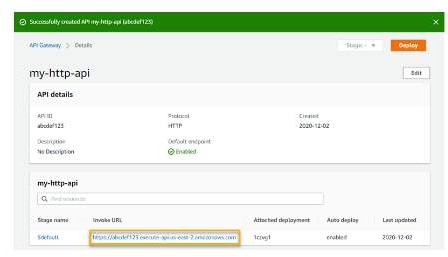
Now you've created an HTTP API with a Lambda integration that's ready to receive requests from clients.

# Step 3: Test your API

Next, you test your API to make sure that it's working. For simplicity, use a web browser to invoke your API.

To test your API

1. Sign in to the API Gateway console at [https://console.aws.amazon.com/apigateway.](https://console.aws.amazon.com/apigateway)
2. Choose your API.
3. Note your API's invoke URL.



1. Copy your API's invoke URL, and enter it in a web browser. Append the name of your Lambda function to your invoke URL to call your Lambda function. By default, the API Gateway console creates a route with the same name as your Lambda function, my-function.

The full URL should look like https://*abcdef123*.execute-api.*us-east- 2*.amazonaws.com/*my-function*.

Your browser sends a GET request to the API.

1. Verify your API's response. You should see the text "Hello from Lambda!" in your browser.

# (Optional) Step 4: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this getting started exercise. The following steps delete your HTTP API, your Lambda function, and associated resources.

To delete an HTTP API

1. Sign in to the API Gateway console at [https://console.aws.amazon.com/apigateway.](https://console.aws.amazon.com/apigateway)
2. On the APIs page, select an API. Choose Actions, and then choose Delete.
3. Choose Delete.

To delete a Lambda function

1. Sign in to the Lambda console at [https://console.aws.amazon.com/lambda.](https://console.aws.amazon.com/lambda)
2. On the Functions page, select a function. Choose Actions, and then choose Delete.
3. Choose Delete.

To delete a Lambda function's log group

1. In the Amazon CloudWatch console, open the [Log groups page.](https://console.aws.amazon.com/cloudwatch/home#logs)
2. On the Log groups page, select the function's log group (/aws/lambda/my-function). Choose Actions, and then choose Delete log group.
3. Choose Delete.

To delete a Lambda function's execution role

1. In the AWS Identity and Access Management console, open the [Roles page](https://console.aws.amazon.com/iam/home?&/roles).
2. Select the function's role, for example, my-function-*31exxmpl*.
3. Choose Delete role.
4. Choose Yes, delete.

You can automate the creation and cleanup of AWS resources by using AWS CloudFormation or AWS SAM. For example AWS CloudFormation templates.