University Of Mumbai
Institute of Distance & Open Learning

PRACTICAL JOURNAL IN PAPER-II

CLOUD APPLICATION DEVELOPMENT

SUBMITTED BY

POOJA HEMANT KAPADIA
APPLICATION ID: 159646
SEAT NO: 0306125

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY PART-II
SEMESTER III

ACADEMIC YEAR
2021-2022

INSTITUTE OF DISTANCE AND OPEN LEARNING
IDOL BUILDING, VIDYANAGARI,
SANTACRUZ (EAST), MUMBAI-400 098

CONDUCTED AT
RIZVI COLLEGE OF ARTS, SCIENCE AND COMMERCE
BANDRA (W), MUMBAI 400050

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

Ms. **POOJA KAPADIA** Application ID: **159646**, Seat No: **0306125** 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 2021-2022.

Section I _____

Section II _____

_____ _____

MSc (IT) Co-ordinator, IDOL External Examiner

TABLE OF CONTENTS

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

2.5 DEVELOP AND DEPLOY APPLICATION OF SERVICE FABRIC

Till now, we have reviewed Service Fabric and its programming models and studied that it can be installed on the cloud or on-premise. Now build some units to better explain that how we produce and use applications on Service Fabric. Here two samples will be cover.

- Situation 1. Express developing an ASP.NET Core stateless web app interacting with an ASP.NET Core stateful API.
- Situation 2. Express developing a Java Spring Boot application adopting Visual Studio Code and use it on Service Fabric as a visitor executable or as a container.

Develop an ASP.NET Core Stateless Web App

We will produce a simple ASP.NET MVC–based application to control operators. The ASP.NET MVC front end communicates with the ASP.NET API to execute CRUD operations. Inside, the Web API utilizes strong groups to save operator data.

Setting up the Development Environment

1. Install Visual Studio 2017.
2. Install the Microsoft Azure Service Fabric SDK.

3. Make assured that the Service Fabric local cluster is running. Secure this by browsing HTTP:// localhost:19080/Explorer/index.html#/ or by right-clicking the Service Fabric icon in the system tray, as shown in Figure 2-7

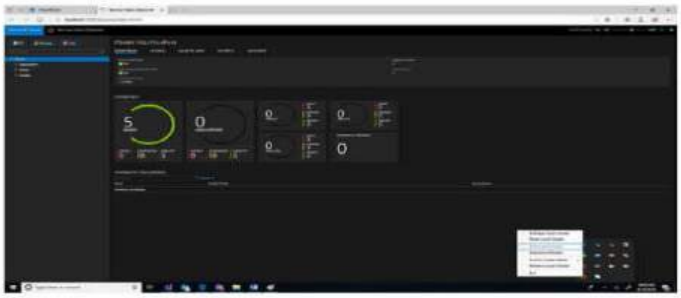


Figure: 2-7. Service Fabric status [3]

Create a ASP.NET Core Web API Using Reliable Collections

Following are the levels.

1. Launch Visual Studio 2017 as an administrator.
2. Create a project by selecting File ➤ New ➤ Project.
3. In the New Project dialog, choose Cloud ➤ Service Fabric Application.
4. Name the Service Fabric application Employee (as depicted in Figure 2-8) and click OK.

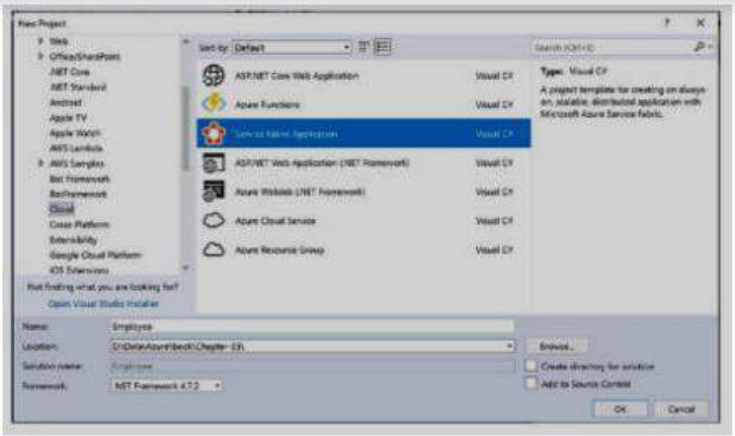


Figure: 2-8. New Service Fabric application [3]

5. Choose Stateful ASP.NET Core, as depicted in Figure 2-9

Figure:2-9. New Stateful ASP.NET Core API [3]

6. You see a screen that looks like Figure 2-10. Click OK

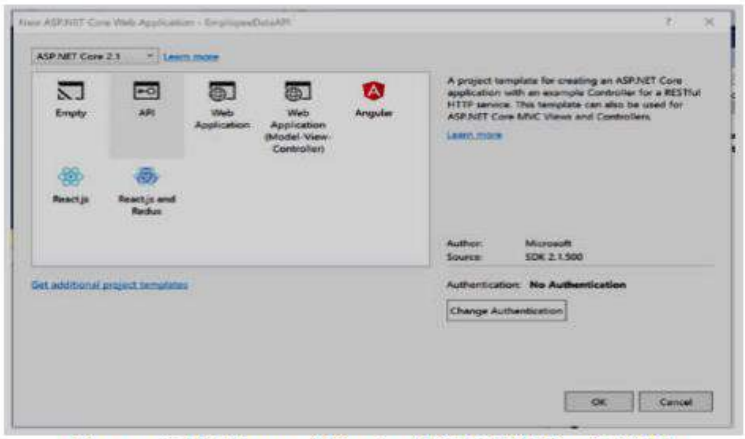


Figure: 2-10. Choose API using (ASP.NET Core 2.1) [3]

7. Right-click the Controller folder in the Employee Data API project and select Add ➤ New Controller, as shown in Figure 2-11. Select API Controller and name the controller Employee Controller

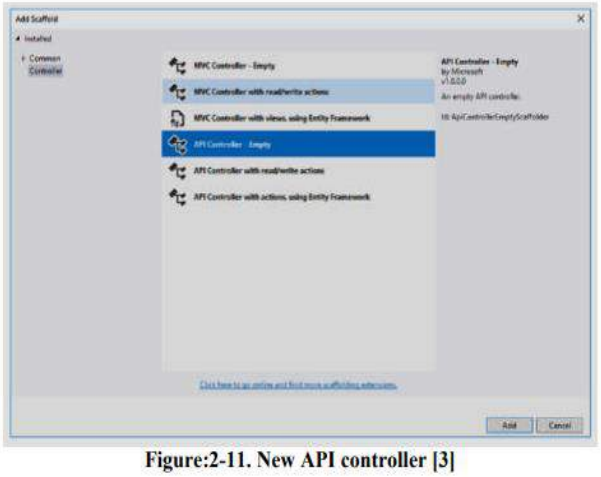


Figure:2-11. New API controller [3]

8. Make sure that the Nu Get packages shown in Figure 2-12 are installed.

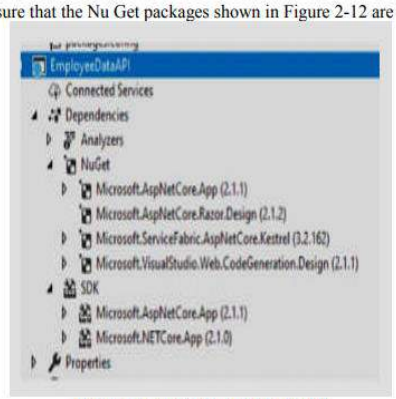


Figure: 2-12. NuGet Packages [3]

9. Replace the file content with the following and compile the changes[3].

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.ServiceFabric.Data;
 using Microsoft.ServiceFabric.Data.Collections;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

```
namespace EmployeeDataAPI.Controllers
{
[Route("api/[controller]")]
[ApiController]
public class EmployeeController : ControllerBase
{
private readonly IReliableStateManager stateManager;
 public EmployeeController(IReliableStateManager stateManager)
{
 this.stateManager = stateManager;
 }
 [HttpGet]
public async Task<ActionResult<List<>>GetAll()
{
 CancellationToken ct = new CancellationToken();
 IReliableDictionary employees = await
this.stateManager.GetOrAddAsync>("employees");
List employeesList = new List();
using (ITransaction tx = this.stateManager. CreateTransaction())
{
```

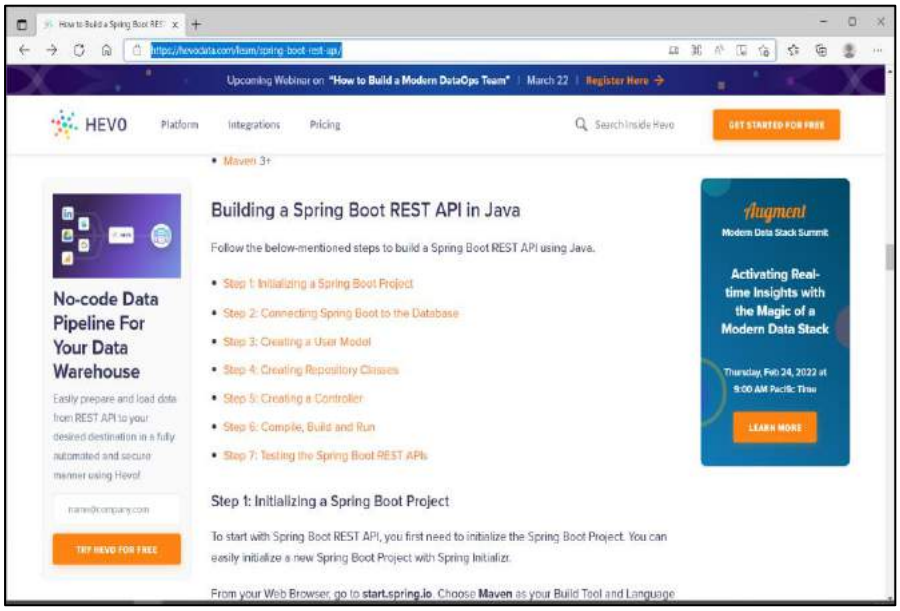
```
{
Microsoft.ServiceFabric.Data.IAsyncEnumerable<KeyValuePair> list =
await employees.CreateEnumerableAsync(tx);
Microsoft.ServiceFabric.Data.IAsyncEnumerator<KeyValuePair>
enumerator = list.GetAsyncEnumerator();
while (await enumerator.MoveNextAsync(ct))
{
employeesList.Add(enumerator.Current.Value);
}
 }
 return new ObjectResult(employeesList);
}
 [HttpGet("{id}")] public async Task<ActionResult>GetEmployee(string
id)
{
 IReliableDictionary employees = await
this.stateManager.GetOrAddAsync>("employees"); Employee employee =
null;
```

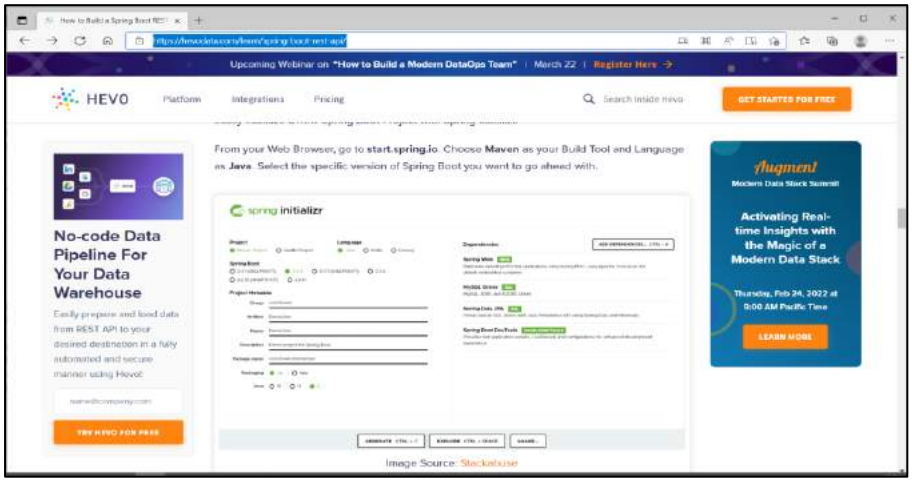
```
using (ITransaction tx = this.stateManager. CreateTransaction())
 {
ConditionalValue currentEmployee = await
employees.TryGetValueAsync(tx, id);
if (currentEmployee.HasValue)
{
employee = currentEmployee.Value;
 }
 }
return new OkObjectResult(employee);
}
[HttpPost]
public async Task Post(Employee employee)
{
IReliableDictionary employees = await
this.stateManager.GetOrAddAsync>("employees"); using (ITransaction tx
= this.stateManager. CreateTransaction())
{
 ConditionalValue currentEmployee = await
employees.TryGetValueAsync(tx, employee. Id.ToString());
 if (currentEmployee.HasValue)
 {
```

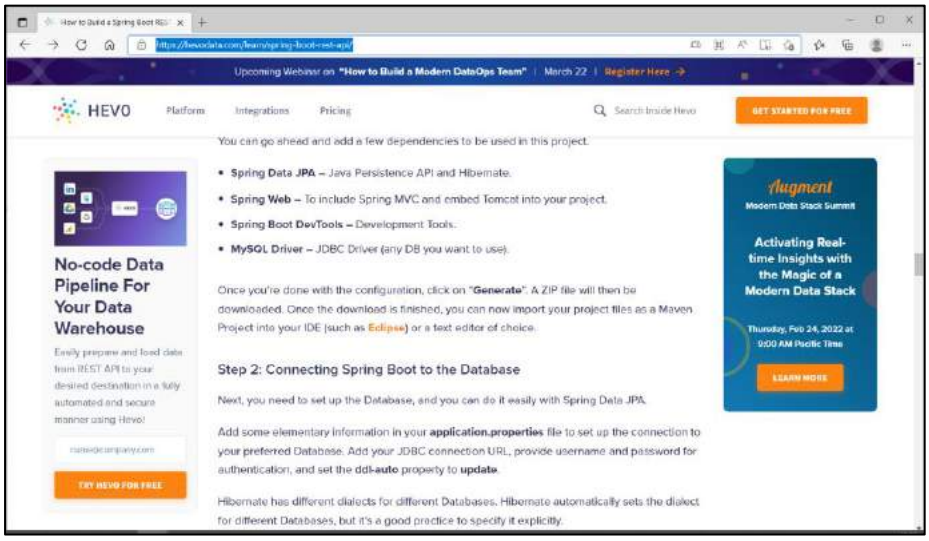
```
 {
await employees.SetAsync(tx, employee. Id.ToString(), employee);
}
Else
 {
 await employees.AddAsync(tx, employee. Id.ToString(), employee);
}
 await tx.CommitAsync();
}
return new OkResult();
}
[HttpDelete("{id}")]
public async Task Delete(string id)
 {
```

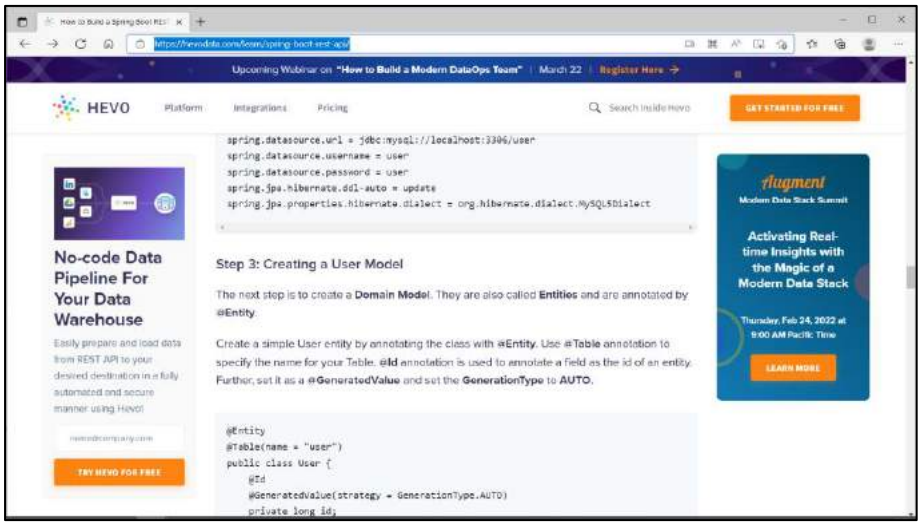
```
IReliableDictionary employees = await
this.stateManager.GetOrAddAsync>("employees"); using (ITransaction tx
= this.stateManager. CreateTransaction())
{
 if (await employees.ContainsKeyAsync(tx, id))
{
 await employees.TryRemoveAsync(tx, id);
 await tx.CommitAsync();
 return new OkResult();
}
else
{
return new NotFoundResult();
}
}
}
 public class Employee
 {
public string Name{ get; set; }
public string Mobile { get; set; }
public long Id { get; set;}
 public string Designation { get; set; }
}
}
```

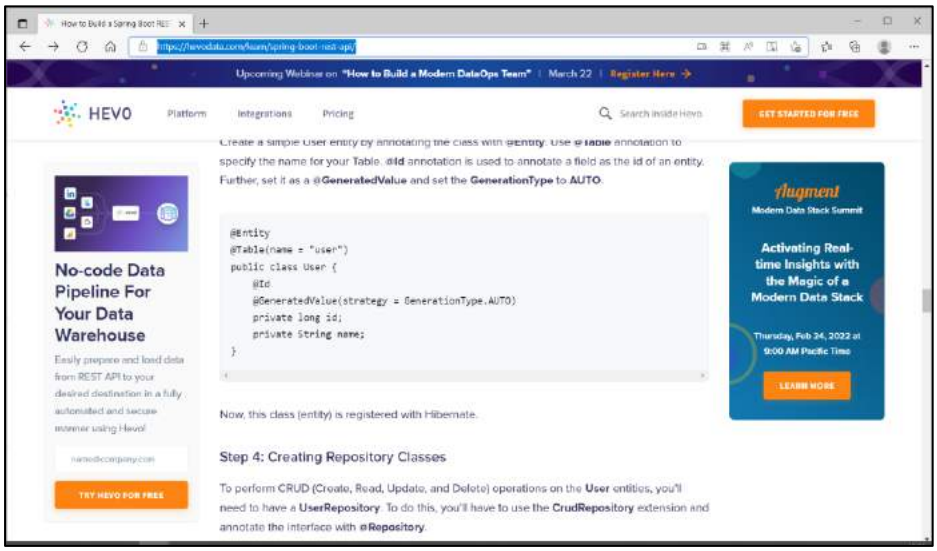
2. Develop a Spring Boot API.

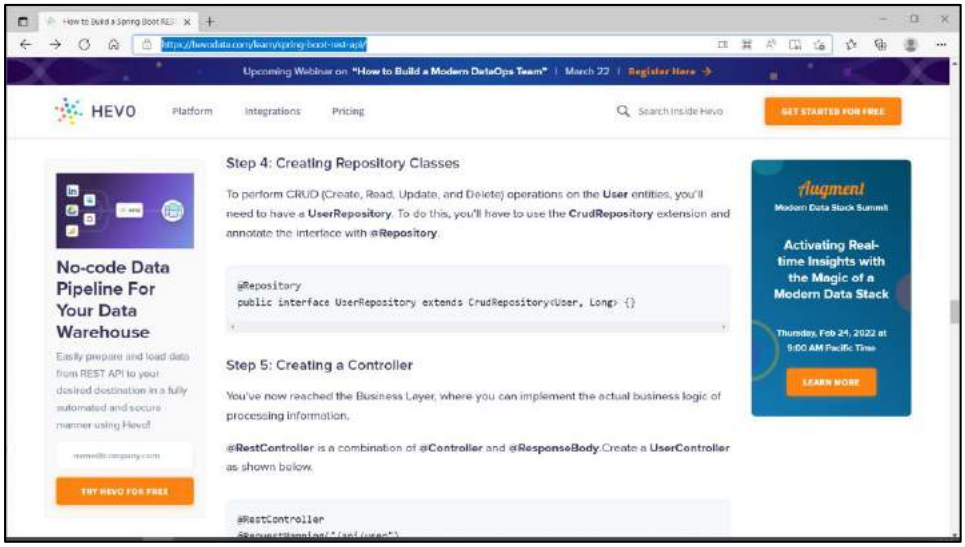


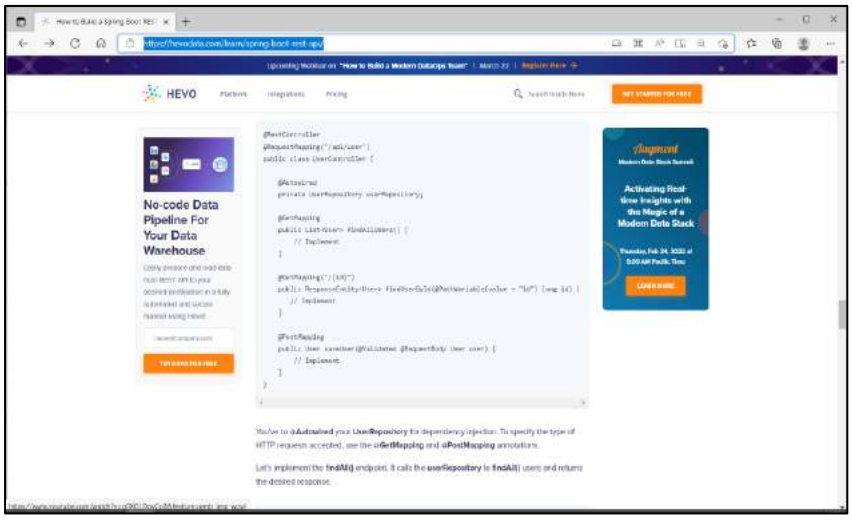


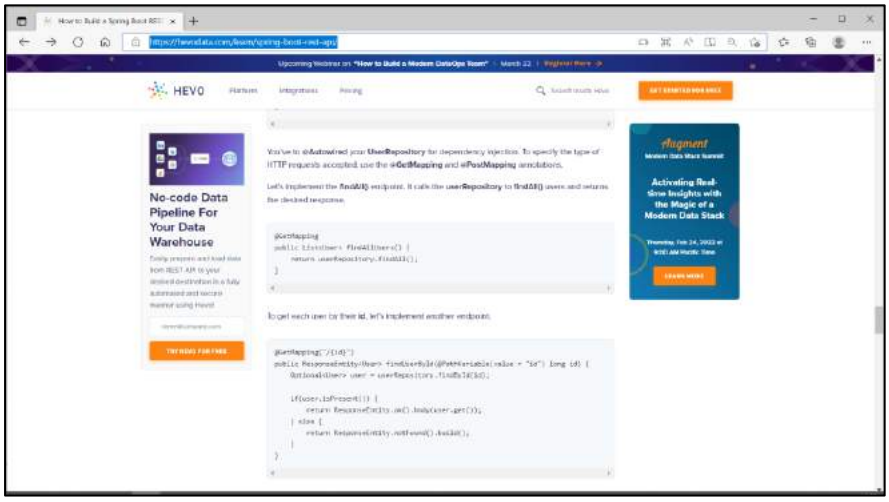


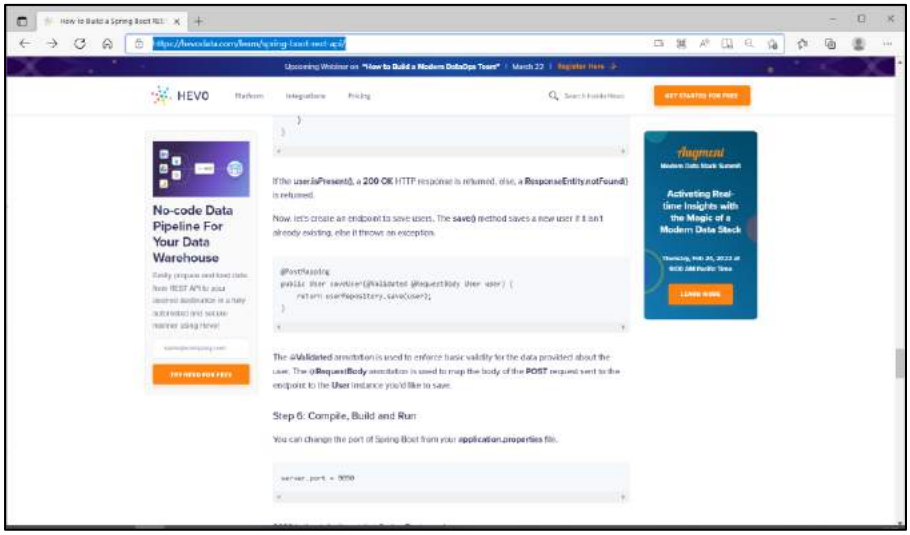

Upcoming Webinar on "How to Build a Modern DataOps Team" | March 22 | Register Here
HEVO
Platform
Integrations
Pricing
GET STARTED FOR FREE
spring.datasource.url = jdbc:mysql://localhost:3306/user
spring.datasource.username = user
spring.datasource.password = user
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
Step 3: Creating a User Model
The next step is to create a Domain Model. They are also called Entities and are annotated by @Entity.
Create a simple User entity by annotating the class with @Entity. Use @Table annotation to specify the name for your Table. @Id annotation is used to annotate a field as the id of an entity. Further, set it as a @GeneratedValue and set the GenerationType to AUTO.
@Entity
@Table(name = "user")
public class User {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
No-code Data Pipeline For Your Data Warehouse
Easily prepare and load data from REST API to your desired destination in a fully automated and secure manner using Hevo!
TRY HEVO FOR FREE
Augment
Modern Data Stack Summit
Activating Real-time Insights with the Magic of a Modern Data Stack
Thursday, Feb 24, 2022 at 9:00 AM Pacific Time
LEARN MORE

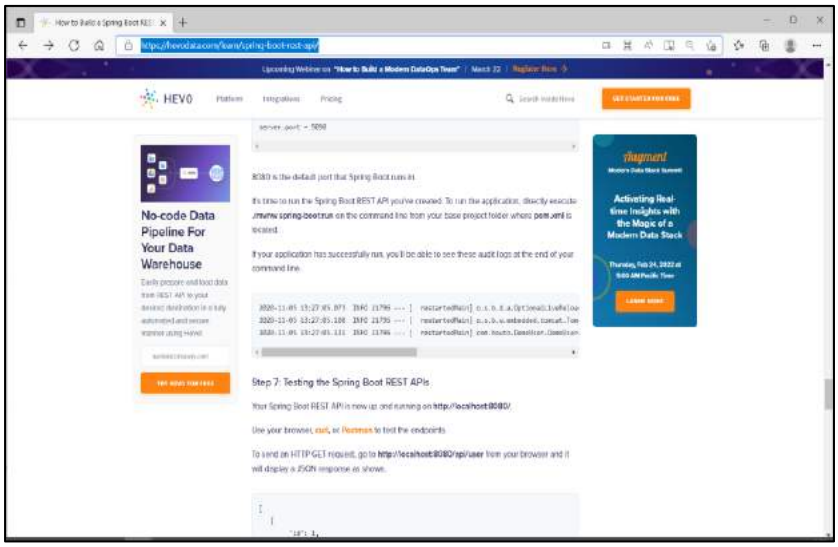

Upcoming Webinar on "How to Build a Modern DataOps Team" | March 22 | Register Here
HEVO
Platform
Integrations
Pricing
GET STARTED FOR FREE
Create a simple User entity by annotating the class with @Entity. Use @Table annotation to specify the name for your Table. @Id annotation is used to annotate a field as the id of an entity. Further, set it as a @GeneratedValue and set the GenerationType to AUTO.
@Entity
@Table(name = "user")
public class User {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
private String name;
}
Now, this class (entity) is registered with Hibernate.
Step 4: Creating Repository Classes
To perform CRUD (Create, Read, Update, and Delete) operations on the User entities, you'll need to have a UserRepository. To do this, you'll have to use the CrudRepository extension and annotate the interface with @Repository.
No-code Data Pipeline For Your Data Warehouse
Easily prepare and load data from REST API to your desired destination in a fully automated and secure manner using Hevo!
TRY HEVO FOR FREE
Augment
Modern Data Stack Summit
Activating Real-time Insights with the Magic of a Modern Data Stack
Thursday, Feb 24, 2022 at 9:00 AM Pacific Time
LEARN MORE

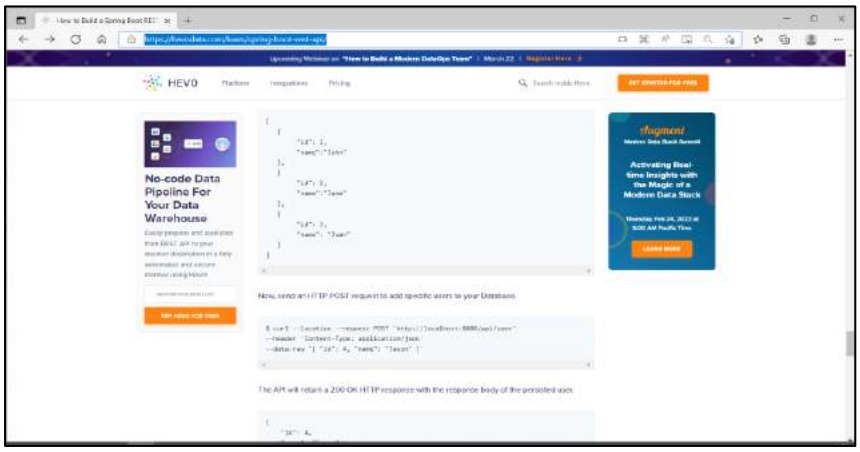

Upcoming Webinar on "How to Build a Modern DataOps Team" | March 22 | Register Here
HEVO
Platform
Integrations
Pricing
GET STARTED FOR FREE
Step 4: Creating Repository Classes
To perform CRUD (Create, Read, Update, and Delete) operations on the User entities, you'll need to have a UserRepository. To do this, you'll have to use the CrudRepository extension and annotate the interface with @Repository.
@Repository
public interface UserRepository extends CrudRepository<User, Long> {}
Step 5: Creating a Controller
You've now reached the Business Layer, where you can implement the actual business logic of processing information.
@RestController is a combination of @Controller and @ResponseBody. Create a UserController as shown below.
@RestController
No-code Data Pipeline For Your Data Warehouse
Easily prepare and load data from REST API to your desired destination in a fully automated and secure manner using Hevo!
TRY HEVO FOR FREE
Augment
Modern Data Stack Summit
Activating Real-time Insights with the Magic of a Modern Data Stack
Thursday, Feb 24, 2022 at 9:00 AM Pacific Time
LEARN MORE

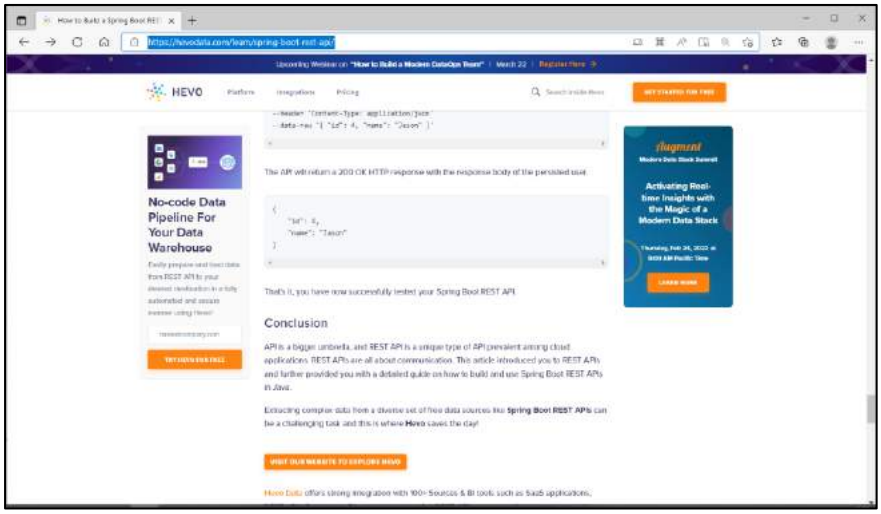

HEVO
No-code Data Pipeline For Your Data Warehouse
Augment
Activating Real-time Insights with the Magic of a Modern Data Stack


HEVO
No-code Data Pipeline For Your Data Warehouse
Augment
Activating Real-time Insights with the Magic of a Modern Data Stack


HEVO
No-code Data Pipeline For Your Data Warehouse
Augment
Activating Real-time Insights with the Magic of a Modern Data Stack


HEVO
No-code Data Pipeline For Your Data Warehouse
Activating Real-time Insights with the Magic of a Modern Data Stack
Step 7: Testing the Spring Boot REST APIs


HEVO
No-code Data Pipeline For Your Data Warehouse
Activating Real-time Insights with the Magic of a Modern Data Stack


HEVO
No-code Data Pipeline For Your Data Warehouse
Activating Real-time Insights with the Magic of a Modern Data Stack
Conclusion

3. 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
2. Configuration Management
3. Secret Management
4. Monitoring & Logging (NLog)
5. Monitoring & Structured Logging (Serilog)
6. Database
7. Documentation
8. CORS
9. Request Validation
10. Global Exception Handling
11. URL Rewriting
12. Deploy .NET API to Azure App Service
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

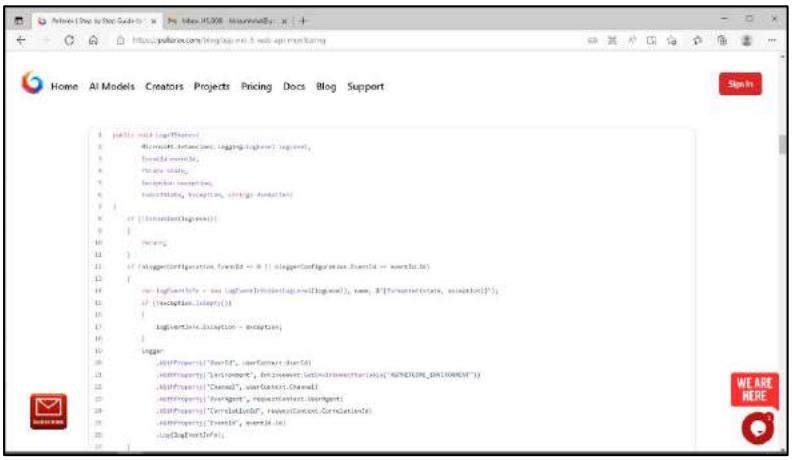
2. Implement Microsoft.Extensions.Logging.ILogger

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.

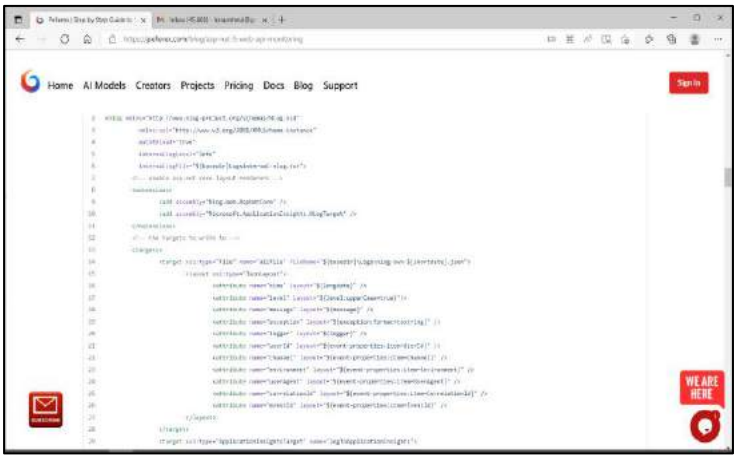
1. Dynamic Log Messages: When we log messages or errors, ideally we should have some dynamic information logged along the main message. Parameters such as UserId, Environment, User Agent, etc. These will help us to troubleshoot problems later on when they happen.

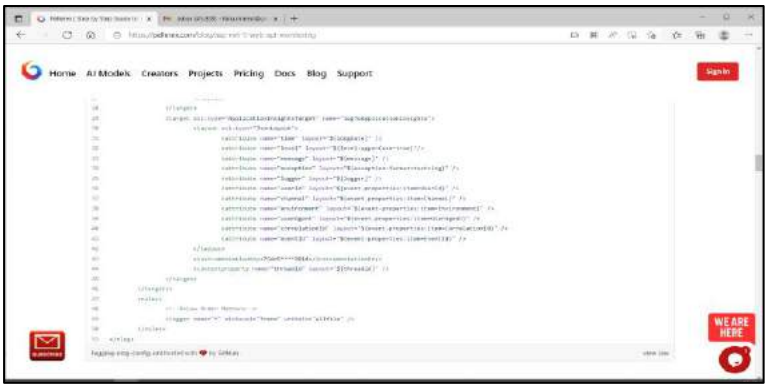
Be mindful of Personally Identifiable Information and Secrets

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.

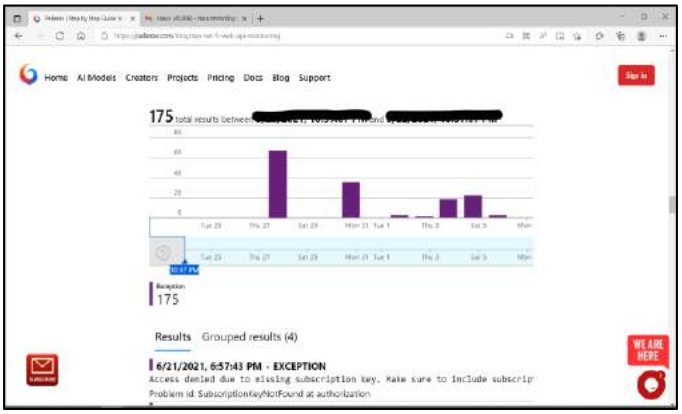
Structured Logging (aka. Semantic Logging)

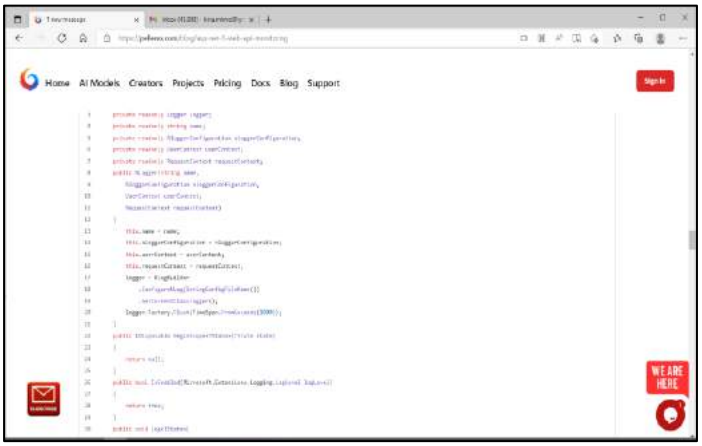
Structured Log format is the ability to separate the log template and log parameters which means logs are not simply converted to text and stored in a text file or a database, which makes it difficult to run query over them. As an example, except if we use some regex to identity certain elements in a 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.

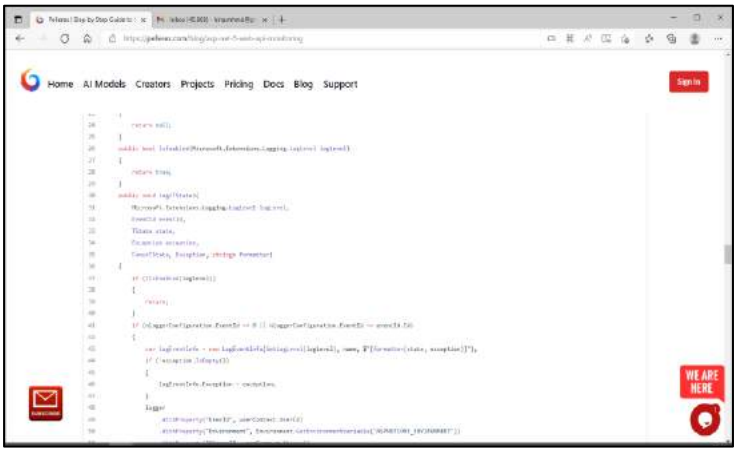
Since version 4.5, NLog has supported Structured Logging beside their event properties. So depending on the target logger and if that target supports structured logging, you can NLog for that purpose.

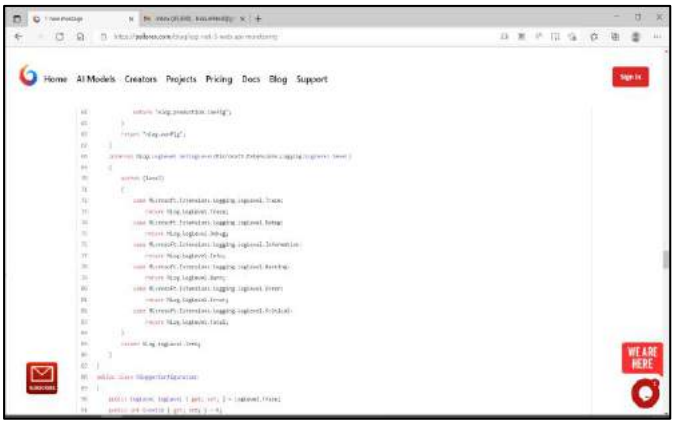
For Pellerex, we have adopted SeriLog, which from the capability point of view, is at the same level as NLog, but comes with a more set of streamlined set of capabilities when it comes down to Structure Logging ream more.

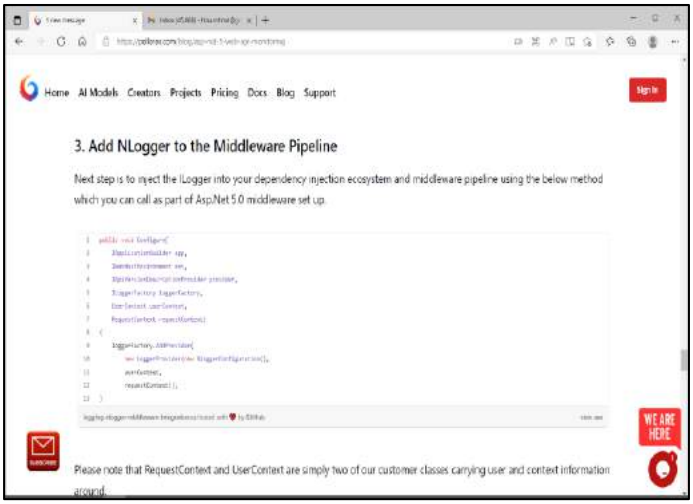
2. Exceptions: In Application Insights, it's important to have log records to be classified properly, information as information, and exception as exception. This is important for troubleshooting, monitoring and analytical purposes. Hence we specifically set the exception of the log if the exception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

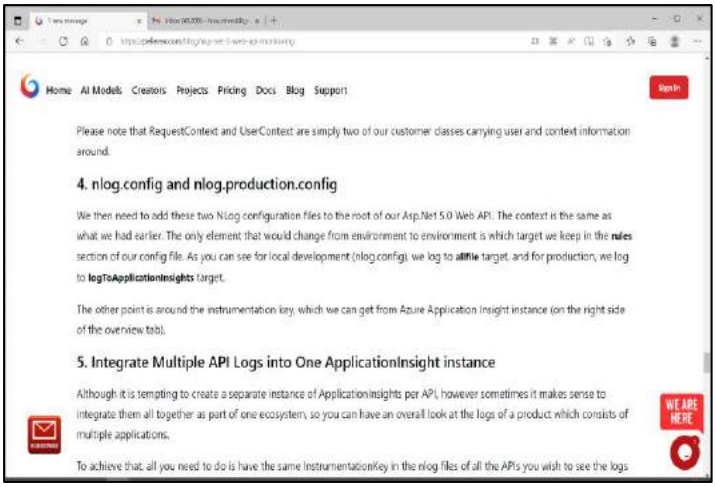


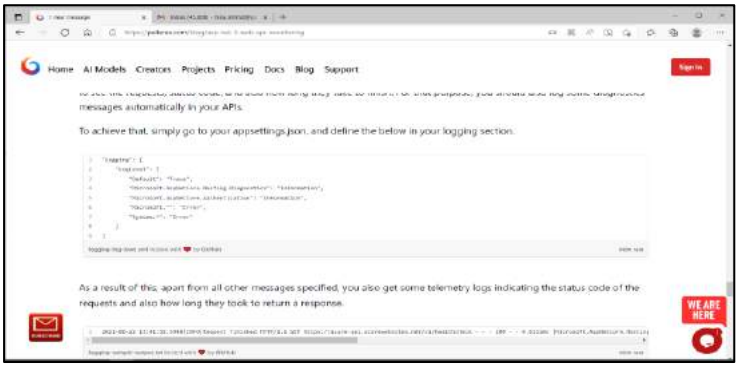




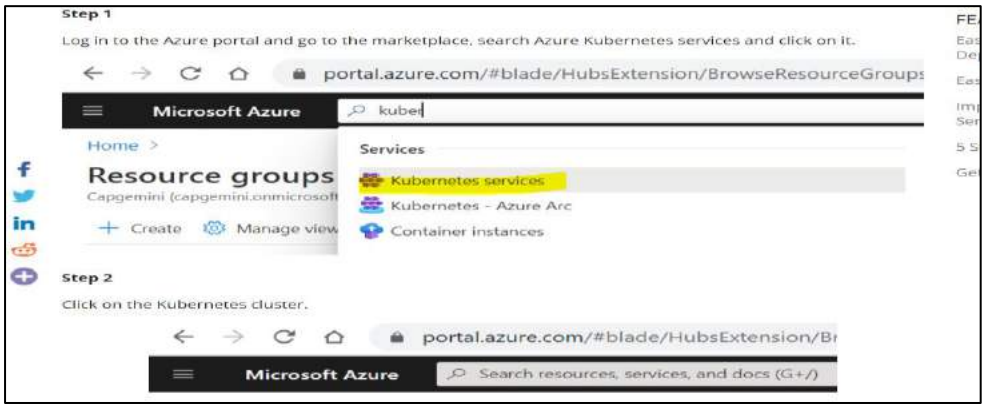


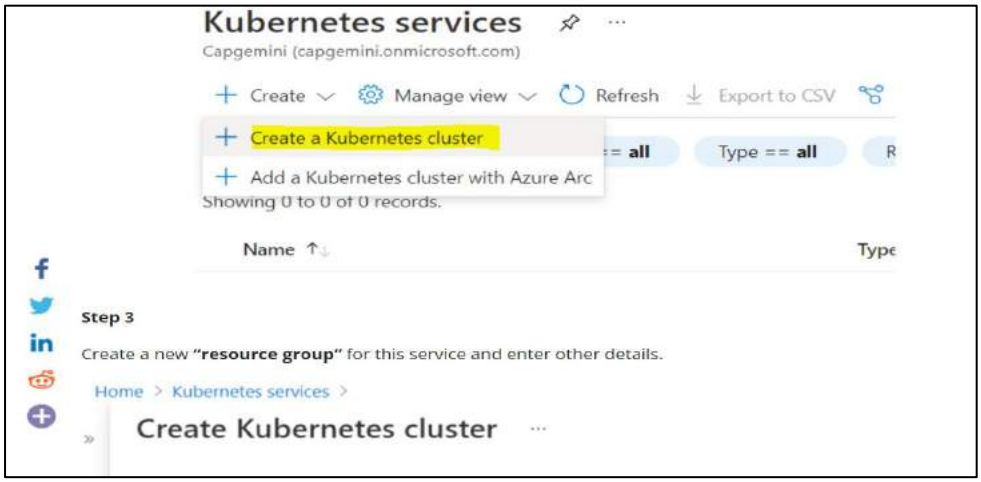

3. Add NLogger to the Middleware Pipeline
Next step is to inject the ILogger into your dependency injection ecosystem and middleware pipeline using the below method which you can call as part of Asp.Net 5.0 middleware set up.
Please note that RequestContext and UserContext are simply two of our customer classes carrying user and context information around.

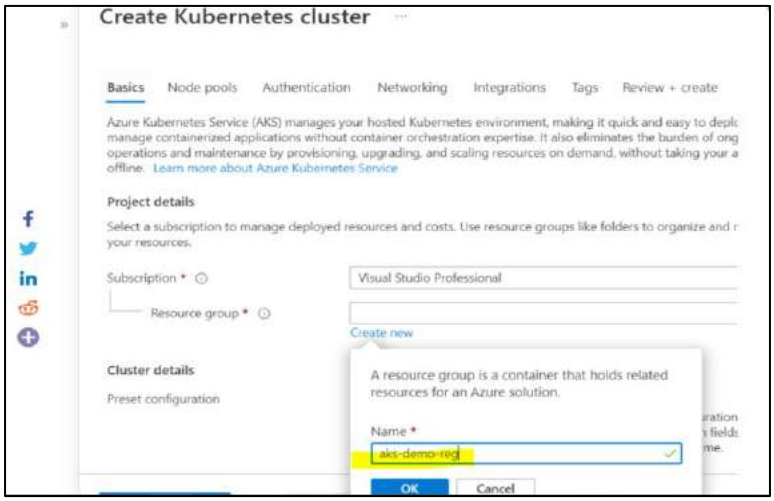

Please note that RequestContext and UserContext are simply two of our customer classes carrying user and context information around.
4. nlog.config and nlog.production.config
We then need to add these two NLog configuration files to the root of our Asp.Net 5.0 Web API. The context is the same as what we had earlier. The only element that would change from environment to environment is which target we keep in the rules section of our config file. As you can see for local development (nlog.config), we log to allfile target, and for production, we log to logToApplicationInsights target.
The other point is around the instrumentation key, which we can get from Azure Application Insight instance (on the right side of the overview tab).
5. Integrate Multiple API Logs into One ApplicationInsight instance
Although it is tempting to create a separate instance of ApplicationInsights per API, however sometimes it makes sense to integrate them all together as part of one ecosystem, so you can have an overall look at the logs of a product which consists of multiple applications.
To achieve that, all you need to do is have the same InstrumentationKey in the nlog files of all the APIs you wish to see the logs

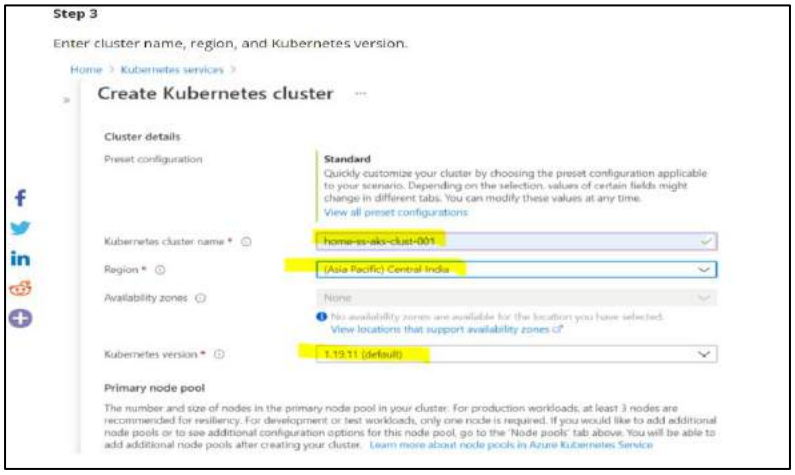

To achieve that, simply go to your appsettings.json, and define the below in your logging section.
As a result of this, apart from all other messages specified, you also get some telemetry logs indicating the status code of the requests and also how long they took to return a response.

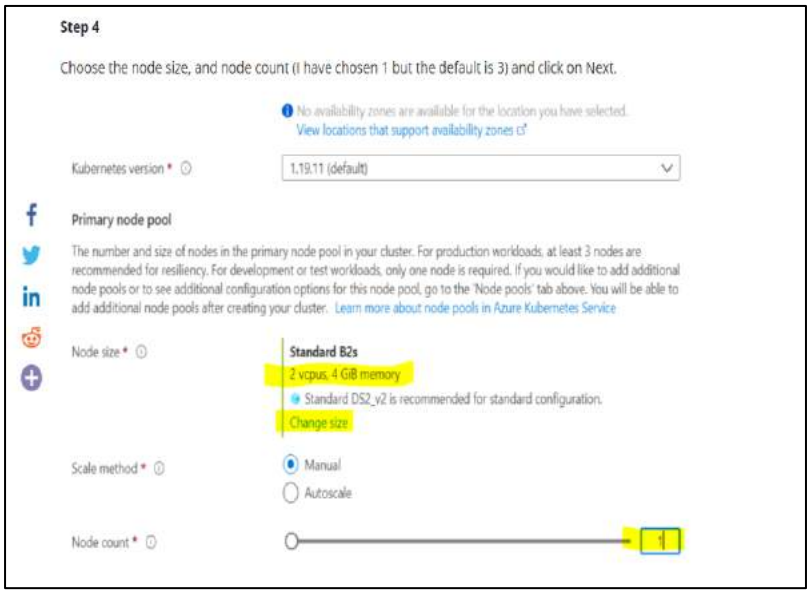
4. a. Create an Azure Kubernetes Service Cluster





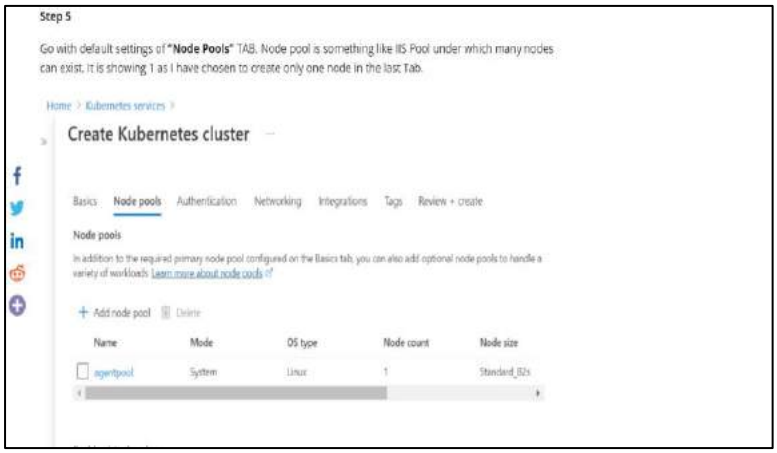

Create Kubernetes cluster
Basics
Node pools
Authentication
Networking
Integrations
Tags
Review + create
Project details
Subscription *
Visual Studio Professional
Resource group *
Create new
Cluster details
Preset configuration
A resource group is a container that holds related resources for an Azure solution.
Name *
aks-demo-rg
OK
Cancel


Step 3
Enter cluster name, region, and Kubernetes version.
Home > Kubernetes services >
Create Kubernetes cluster
Cluster details
Preset configuration
Standard
Quickly customize your cluster by choosing the preset configuration applicable to your scenario. Depending on the selection, values of certain fields might change in different tabs. You can modify these values at any time.
View all preset configurations
Kubernetes cluster name *
home-rs-aks-clust-001
Region *
(Asia Pacific) Central India
Availability zones
None
Kubernetes version *
1.19.11 (default)
Primary node pool
The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. Learn more about node pools in Azure Kubernetes Service


Step 4
Choose the node size, and node count (I have chosen 1 but the default is 3) and click on Next.
No availability zones are available for the location you have selected.
View locations that support availability zones
Kubernetes version *
1.19.11 (default)
Primary node pool
The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. Learn more about node pools in Azure Kubernetes Service
Node size *
Standard B2s
2 vcpus, 4 GiB memory
Standard DS2_v2 is recommended for standard configuration.
Change size
Scale method *
Manual
Autoscale
Node count *
1

Step 5

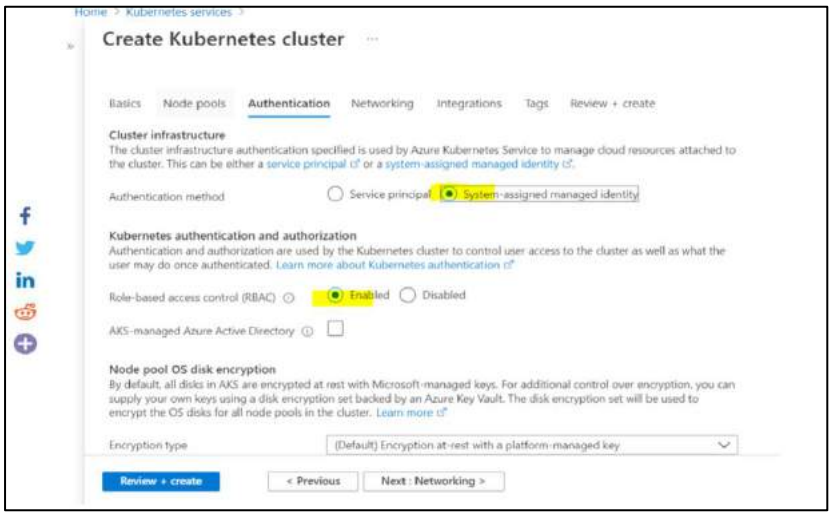
Go with default settings of **"Node Pools"** TAB. Node pool is something like IIS Pool under which many nodes can exist. It is showing 1 as I have chosen to create only one node in the last Tab.



Step 6

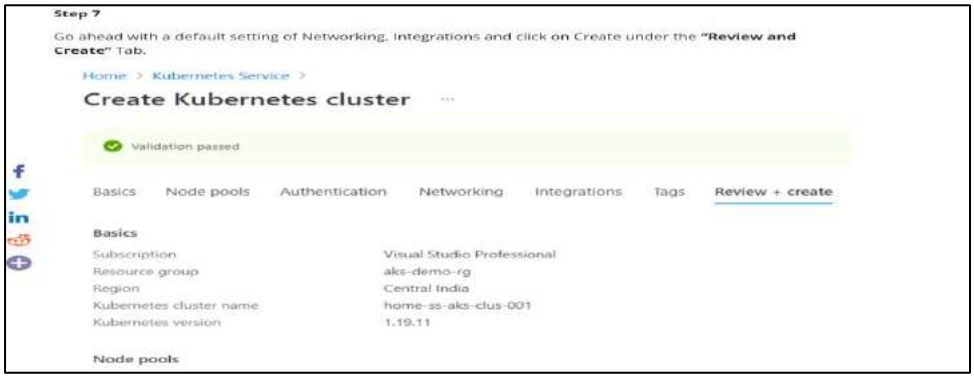
Go ahead with the default settings of the **"Authentication"** Tab.

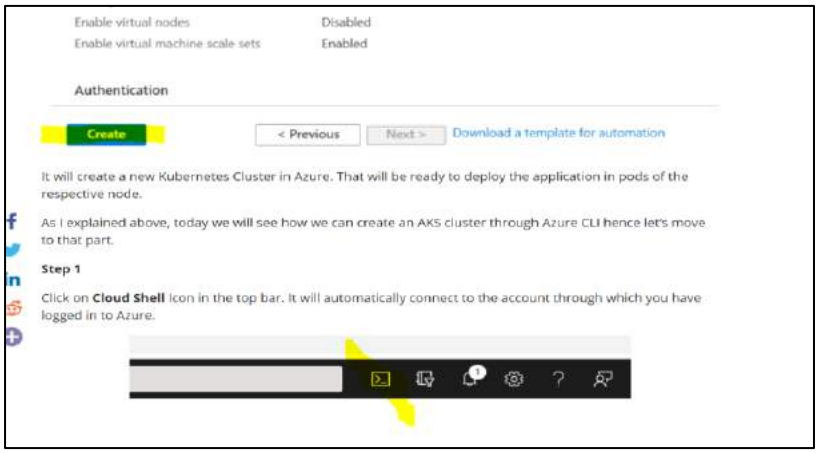
Note
The system-assigned managed identity authentication method must be used in order to associate an Azure Container Registry.



Step 7

Go ahead with a default setting of Networking, Integrations and click on Create under the **"Review and Create"** Tab.





It will create a new Kubernetes Cluster in Azure. That will be ready to deploy the application in pods of the respective node.

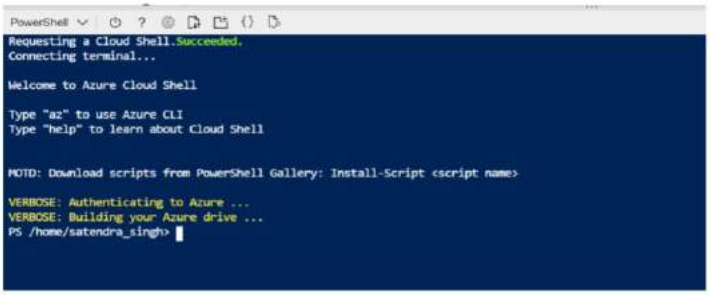
As I explained above, today we will see how we can create an AKS cluster through Azure CLI hence let's move to that part.

Step 1

Click on **Cloud Shell** icon in the top bar. It will automatically connect to the account through which you have logged in to Azure.

Step 2

The next screen will be shown as below.



Step 3

Now execute the below command, to create an AKS Cluster. It will take 5-10 min to complete the execution.

Step 3

Now execute the below command, to create an AKS Cluster. It will take 5-10 min to complete the execution.

Note

Ensure to generate the SSH keys. This can allow us to log into the cluster if required.

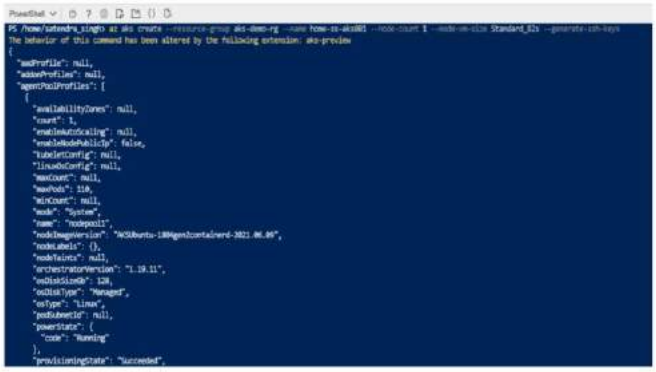
az aks create --resource-group aks-demo-rg --name home-ss-aks001 --node-count 1 --node-vm-size Standard_B2s--generate-ssh-keys

Step 4

Once the execution is completed, it will give the complete output in JSON format. You can go through that if required.

Step 4

Once the execution is completed, it will give the complete output in JSON format. You can go through that if required.



Step 5

Now go to the portal and review the resources created under our resource group **"aks-demo-rg"**.

Note

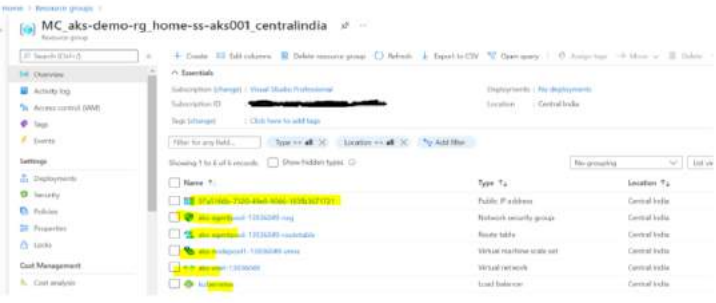
When an AKS cluster is created, some other resources are also created along with it, under a different resource group called **"MC_aks-demo-rg_home-ss-aks001_centralindia"**

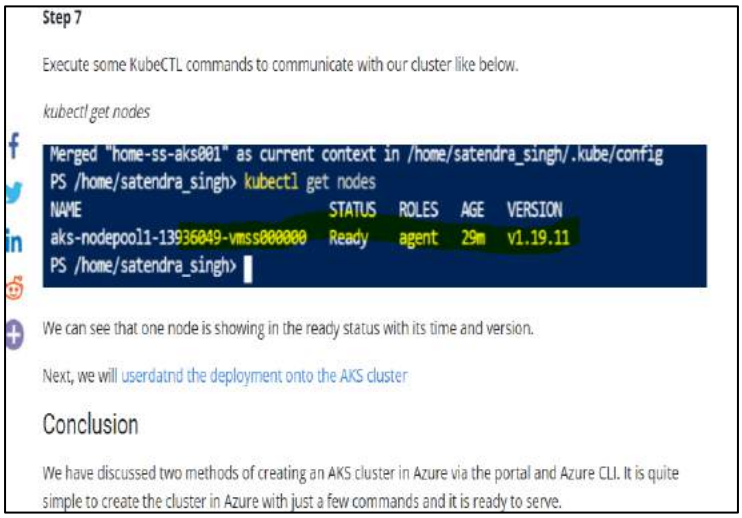
I will write another article about these resources, to explain in detail.

Note

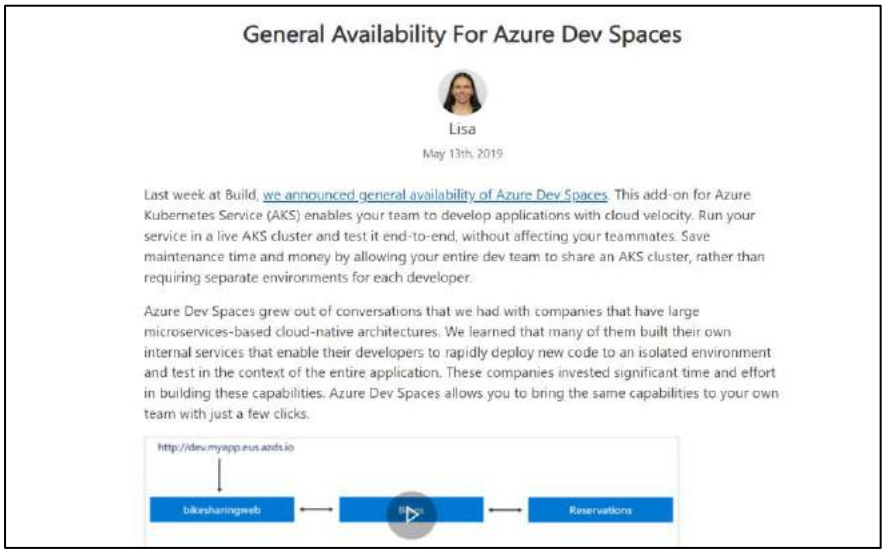
When an AKS cluster is created, some other resources are also created along with it, under a different resource group called **"MC_aks-demo-rg_home-ss-aks001_centralindia"**

I will write another article about these resources, to explain in detail.





b. Enable Azure Dev Spaces on an AKS Cluster



In this post, we will show you how to get started with Azure Dev Spaces:

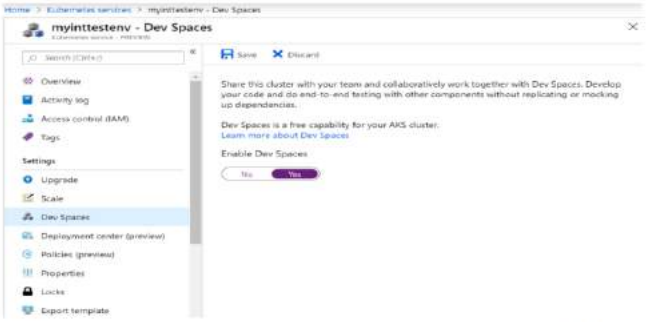
1. If you're a team lead or devops, you'll learn how to set up your team's AKS cluster for use with Azure Dev Spaces.
2. If you're a developer, you'll learn how to run a service inside your team's shared AKS cluster.
3. You'll also learn how to troubleshoot a bug using the Dev Spaces extension for VS Code. (An extension for Visual Studio is also available.)

Setting up your AKS cluster

Let's say that you are running all the services that make up your application in an AKS cluster that serves as your team's integration testing environment.

```
$ kubectl get services --namespace dev
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE
bikes               ClusterIP      10.0.8.84      <none>           80/TCP           2m
bikesharingweb      ClusterIP      10.0.137.26    <none>           80/TCP           2m
billing             ClusterIP      10.0.216.16    <none>           80/TCP           2m
databases-mongo     ClusterIP      10.0.109.152   <none>           27017/TCP        2m
databases-sql       ClusterIP      10.0.141.173   <none>           1433/TCP         2m
gateway             LoadBalancer   10.0.234.116   52.229.49.251    80:30470/TCP     2m
populatedatabase    ClusterIP      10.0.121.116   <none>           80/TCP           2m
reservation         ClusterIP      10.0.200.230   <none>           80/TCP           2m
reservationengine   ClusterIP      10.0.77.9      <none>           80/TCP           2m
users               ClusterIP      10.0.112.52    <none>           80/TCP           2m
```

You can enable Dev Spaces on the cluster from the Azure portal or the Azure CLI. The screen below shows where to enable Dev Spaces in the Azure portal.



Then, you can configure the namespace where the services are running as a *dev space*, which enables Dev Spaces functionality.

Then, you can configure the namespace where the services are running as a *dev space*, which enables Dev Spaces functionality.

```
$ azds space select -n dev
Dev space 'dev' does not exist and will be created.

Select a parent dev space or Kubernetes namespace to use as a parent dev space.
 [0] <none>
 [1] default
Type a number: 0

Creating and selecting dev space 'dev'...2s
```

Now that you've set up the cluster and the application properly, let's see how individual developers on your team can test their code in the context of the full application using Dev Spaces.

Running a service in AKS

Suppose that a new developer named Jane has joined your team. You have a new feature that you want her to create inside an existing microservice called *Bikes*.

Traditionally, Jane would write the code for the feature on her local development workstation and do some basic validation of the feature by running the *Bikes* service locally. Hopefully your team has already invested in some automated integration tests that she can run to further validate that she hasn't broken anything. But since she's new to the application and its codebase, she might not feel confident checking in her code until she's seen it working properly in the context of the full application. Automated tests can't catch everything, and no one wants to break the team's dev environment, especially on their first day on the team!

This is where Azure Dev Spaces can make Jane's first check-in experience easy and positive.

Jane can create a *child dev space* called *newfeature*. The parent of *newfeature* is the dev space you configured when you initially set up Dev Spaces for your team, which is running the entire application.

```
$ azds space select -n newfeature
Dev space 'newfeature' does not exist and will be created.

Select a parent dev space or Kubernetes namespace to use as a parent dev space.
 [0] <none>
 [1] default
 [2] dev
Type a number: 2

Creating and selecting dev space 'dev/newfeature'...3s
```

The version of the application that runs in the child dev space has its own URL. This is simply the URL to the team's version of the application, prefixed with *newfeature.s*. Azure Dev Spaces intercepts requests that come in with this URL prefix and routes them appropriately. If there is a version of the service running in the *newfeature* dev space, then Dev Spaces routes the request to that version. Otherwise, Dev Spaces routes the request to the team's version of the service, running in the root dev space.

End-to-End Testing

Jane can leverage this functionality to quickly test her changes end-to-end, even before she checks in her code. All she needs to do is run her updated version of the Bikes service in the *newfeature* dev space. Now she can access her version of the application by using the *newfeature.s* URL. Azure Dev Spaces will automatically handle routing requests between Jane's updated version of Bikes (running in the *newfeature* dev space) and the rest of the services that make up the application (running in the parent dev space).

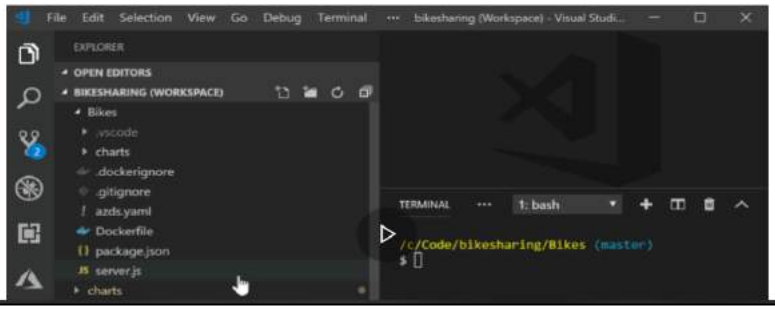
In the example shown below, the site currently shows a generic bicycle icon for each listed bicycle. One of Jane's teammates has updated the database to include a picture of the actual bicycle. Jane needs to update the Bikes service to pull this picture from the database and send it along to the upstream services:

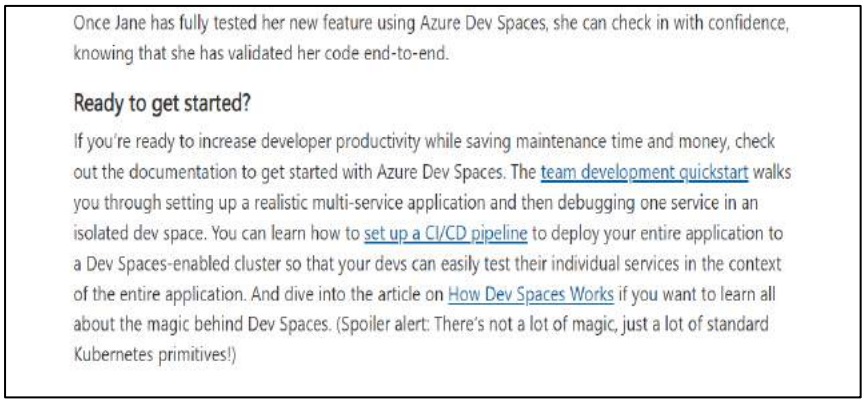
Troubleshooting a bug using Azure Dev Spaces

What if Jane discovers her changes didn't work properly? First of all, her broken code is only running in her *newfeature* dev space. Her teammates' requests still use the original version of Bikes running in the parent dev space. She can take her time troubleshooting the problem, knowing that she's not blocking her teammates.

In addition, she can use the Azure Dev Spaces extensions for Visual Studio or Visual Studio Code to debug her code running live in the cloud with a single click. This allows her to quickly zero in on the problem, fix it, and validate her fix. She can even run and debug additional services inside the *newfeature* dev space, if the problem spans multiple services.

The following video shows debugging a Node.js service through VS Code, but the same capabilities are available for .NET Core and Java, inside Visual Studio or VS Code:



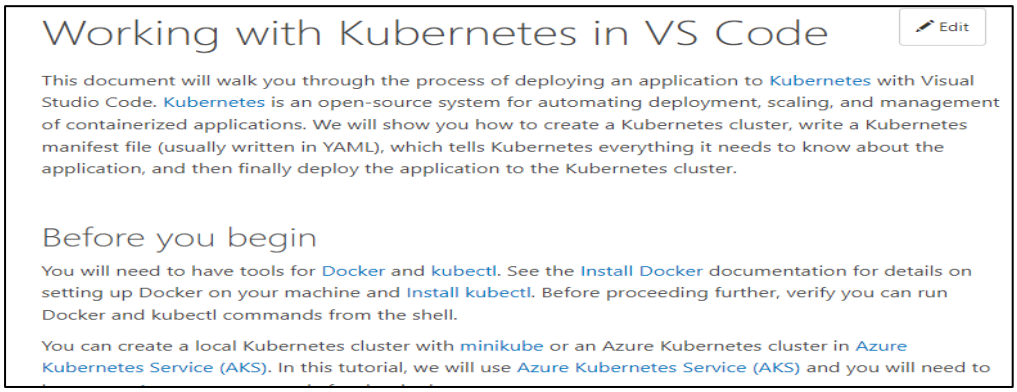


Once Jane has fully tested her new feature using Azure Dev Spaces, she can check in with confidence, knowing that she has validated her code end-to-end.

Ready to get started?

If you're ready to increase developer productivity while saving maintenance time and money, check out the documentation to get started with Azure Dev Spaces. The team development quickstart walks you through setting up a realistic multi-service application and then debugging one service in an isolated dev space. You can learn how to set up a CI/CD pipeline to deploy your entire application to a Dev Spaces-enabled cluster so that your devs can easily test their individual services in the context of the entire application. And dive into the article on How Dev Spaces Works if you want to learn all about the magic behind Dev Spaces. (Spoiler alert: There's not a lot of magic, just a lot of standard Kubernetes primitives!)

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



Working with Kubernetes in VS Code

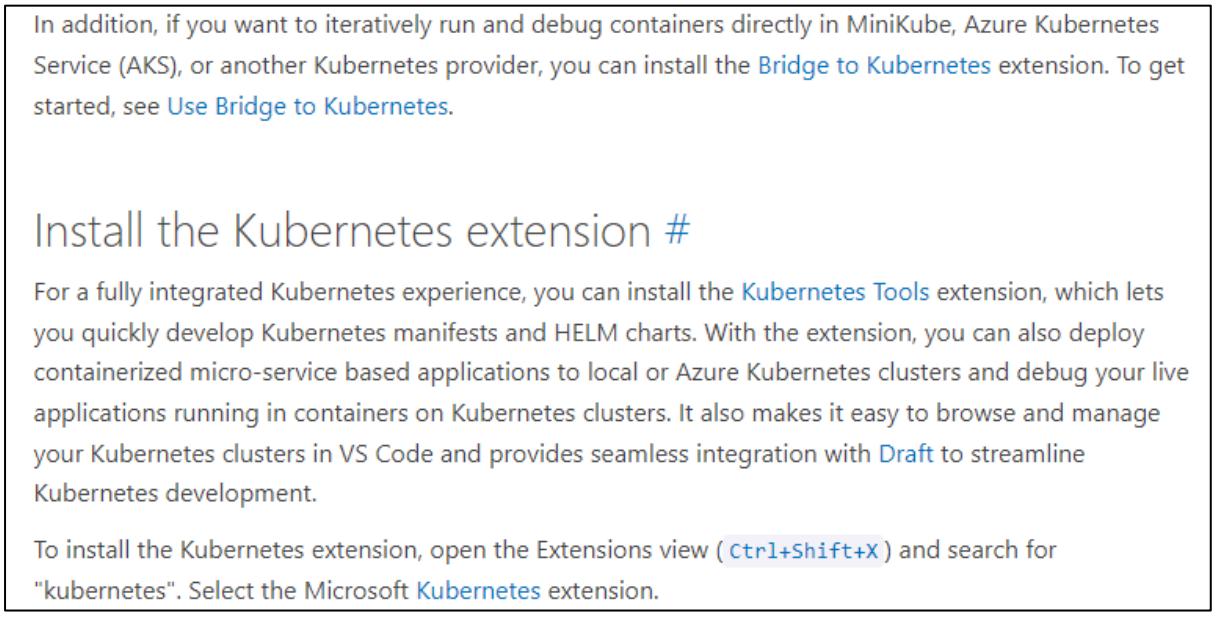
Edit

This document will walk you through the process of deploying an application to Kubernetes with Visual Studio Code. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. We will show you how to create a Kubernetes cluster, write a Kubernetes manifest file (usually written in YAML), which tells Kubernetes everything it needs to know about the application, and then finally deploy the application to the Kubernetes cluster.

Before you begin

You will need to have tools for Docker and kubectl. See the Install Docker documentation for details on setting up Docker on your machine and Install kubectl. Before proceeding further, verify you can run Docker and kubectl commands from the shell.

You can create a local Kubernetes cluster with minikube or an Azure Kubernetes cluster in Azure Kubernetes Service (AKS). In this tutorial, we will use Azure Kubernetes Service (AKS) and you will need to

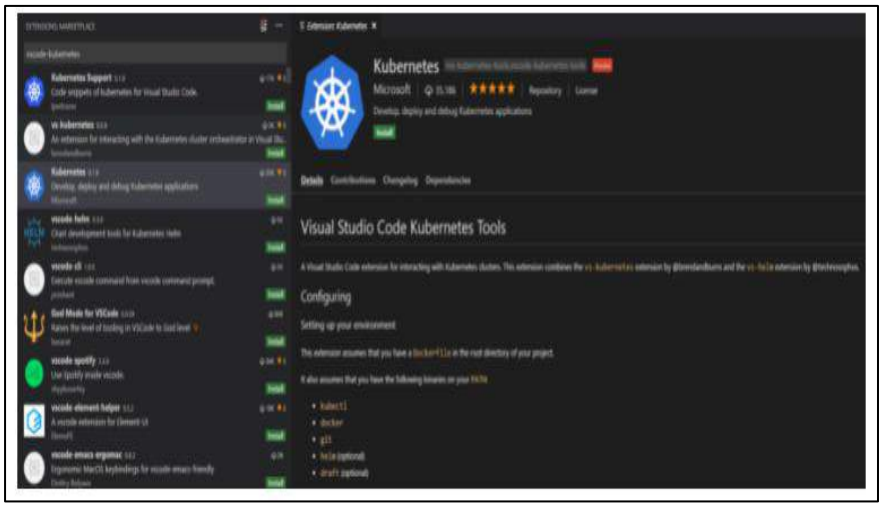


In addition, if you want to iteratively run and debug containers directly in MiniKube, Azure Kubernetes Service (AKS), or another Kubernetes provider, you can install the Bridge to Kubernetes extension. To get started, see Use Bridge to Kubernetes.

Install the Kubernetes extension

For a fully integrated Kubernetes experience, you can install the Kubernetes Tools extension, which lets you quickly develop Kubernetes manifests and HELM charts. With the extension, you can also deploy containerized micro-service based applications to local or Azure Kubernetes clusters and debug your live applications running in containers on Kubernetes clusters. It also makes it easy to browse and manage your Kubernetes clusters in VS Code and provides seamless integration with Draft to streamline Kubernetes development.

To install the Kubernetes extension, open the Extensions view (`Ctrl+Shift+X`) and search for "kubernetes". Select the Microsoft Kubernetes extension.

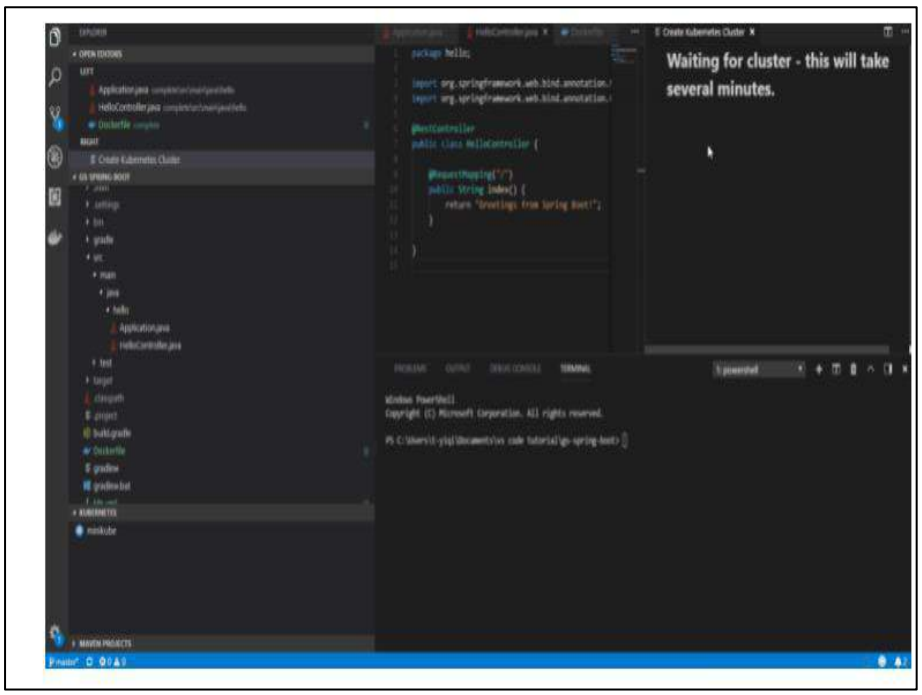


Containerize and publish the application

You can follow the Working with Docker tutorial to build your project, generate a Docker image, and push it to a public or private container registry through the Microsoft Docker Extension.

Create and config a Kubernetes cluster

You can create a Kubernetes cluster running on Azure using the Kubernetes extension in VS Code. Once you have installed the Kubernetes extension, you will see **KUBERNETES** in the Explorer. Click on **More** and choose **Create Cluster**. Follow the instructions to choose the cluster type (here we choose **Azure Kubernetes Service**), select your subscription, and set up the Azure cluster and Azure agent settings. It will take a few minutes to complete the whole workflow.



Important: To create a Kubernetes cluster on Azure, you need to install the Azure CLI and sign in.

Tip: You will encounter an error if you don't have an available RSA key file. Follow create SSH public-private key to create your key before creating an Azure Kubernetes cluster.

Error creating cluster

An error occurred while creating the cluster.

Details

ERROR: An RSA key file or key value must be supplied to SSH Key Value. You can use --generate-ssh-keys to let CLI generate one for you

Error creating cluster

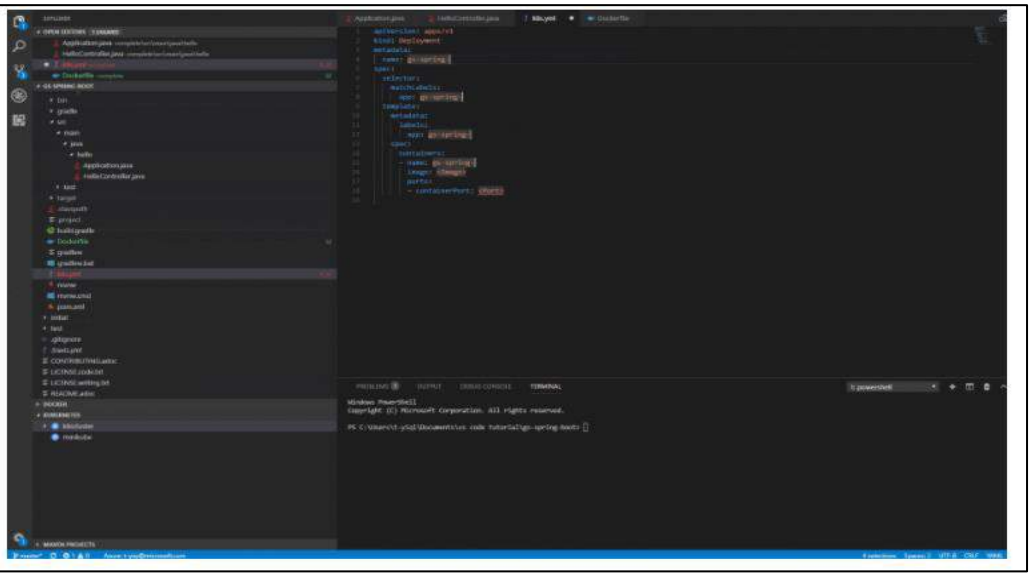
An error occurred while creating the cluster.

Details

ERROR: Operation failed with status: 'Bad Request'. Details: The VM size of Agent is not allowed in your subscription in location 'centralus'. Agent VM size 'Standard_D2_v2' is available in locations: centraluseuap,eastus,eastus2euap,southcentralus,southeastasia,westcentralus,westeurope,westus2.

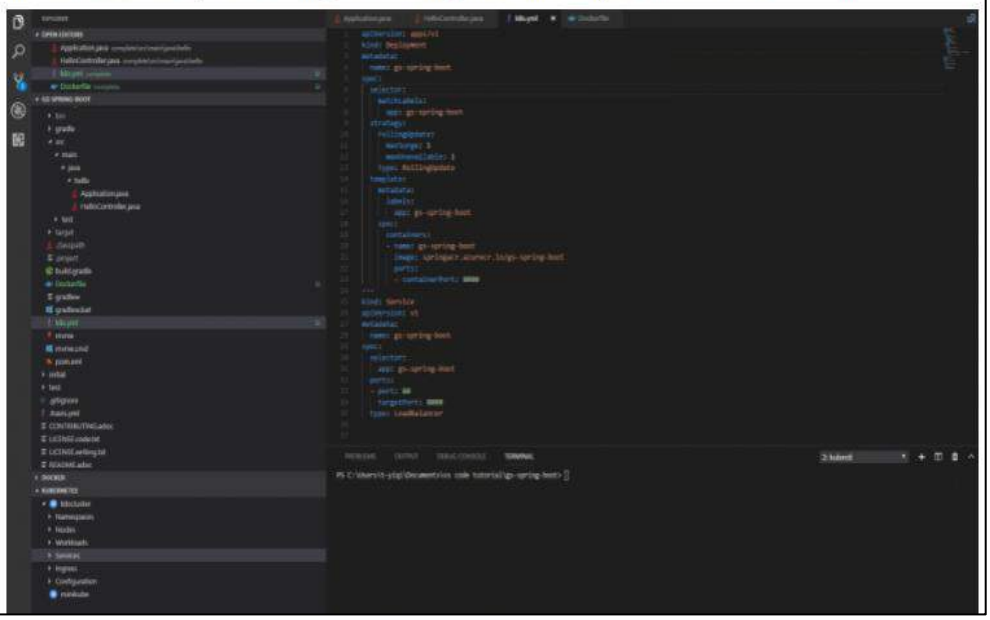
Deploy the application to Azure Kubernetes Service

The Kubernetes extension provides autocompletion, code snippets, and verification for the Kubernetes manifest file. For example, once you type 'Deployment' in an empty YAML file, a manifest file with fundamental structure is autogenerated for you. You only need to enter your app name, image, and port manually.



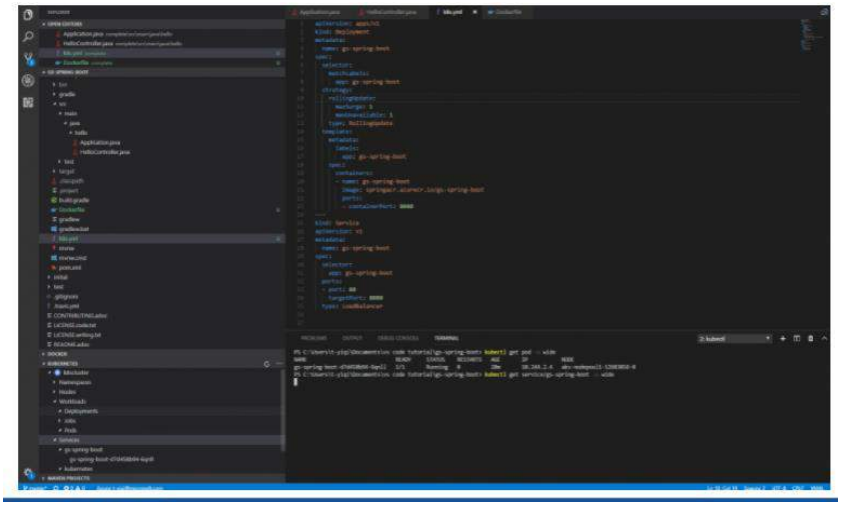
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gs-spring-boot
spec:
  selector:
    matchLabels:
      app: gs-spring-boot
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: gs-spring-boot
    spec:
      containers:
      - name: gs-spring-boot
        image: springacr.azurecr.io/gs-spring-boot
        ports:
        - containerPort: 8080
---
kind: Service
apiVersion: v1
metadata:
  name: gs-spring-boot
spec:
  selector:
    app: gs-spring-boot
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
```

Once your manifest file is ready, you only need one command to start a deployment. Open the **Command Palette** (`Ctrl+Shift+P`) and run **Kubernetes: Create**. It will deploy the application to your Kubernetes cluster and create objects according to the configuration in the open Kubernetes manifest file.



Checking on your deployment

After deployment, the Kubernetes extension can help you check the status of your application. From the Explorer, click on **Workloads**, right click on **Pods** and then choose **Get** to see whether the application has started. To view the status of your app, select **Services**, right click on your app, and then click **Get**. The status will be printed to the Integrated Terminal. Once your application has an `EXTERNAL_IP`, you can open a browser and see your web app running.

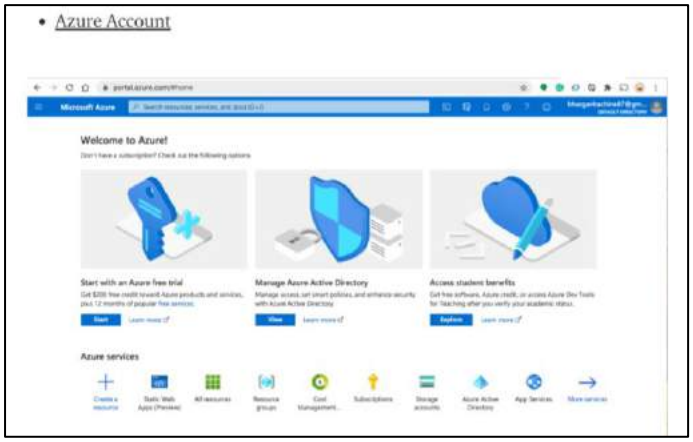


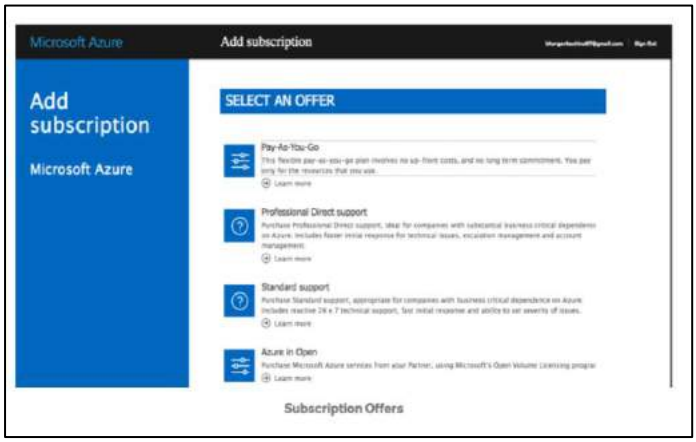
d. Configure Visual Studio Code to Work with an Azure Kubernetes Service Cluster

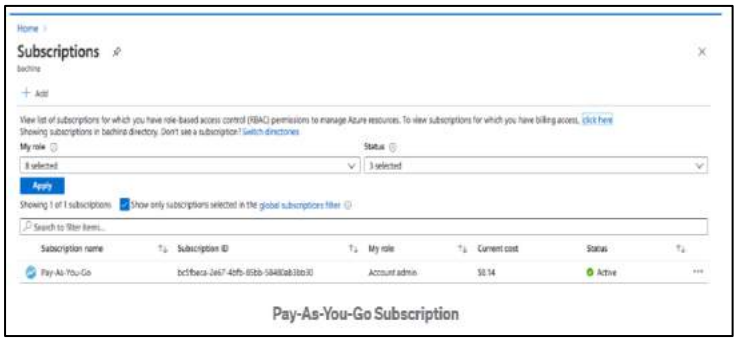
e. Deploy Application on AKS

i. Core Web API

ii. Node.js API







Pay-As-You-Go Subscription

Install Azure CLI and Configure

```
Bhargavs-MacBook-Pro:sample-workspace bhargavbachina$ az login
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "44e2e07f-a19d-42b6-80a9-a8e4097f3948",
    "id": "bc5fbeca-2e67-4bfb-85bb-58480ab3bb30",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Pay-As-You-Go",
    "state": "Enabled",
    "tenantId": "44e2e07f-a19d-42b6-80a9-a8e4097f3948",
    "user": {
      "name": "bhargavbachina87@gmail.com",
      "type": "user"
    }
  }
]
```

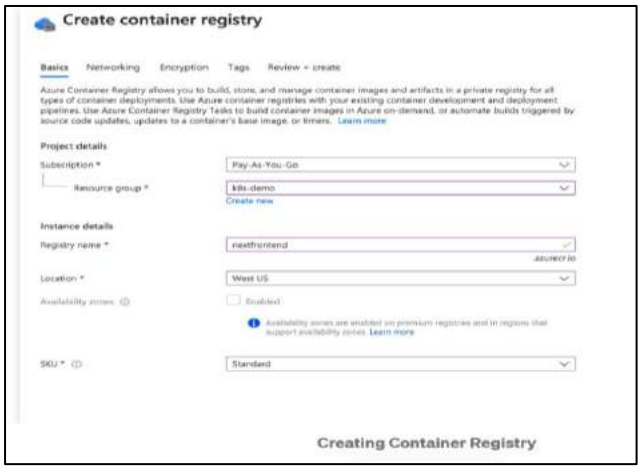
Dockerize the Project

```
1  FROM node:10 AS ui-build
2  WORKDIR /usr/src/app
3  COPY my-app/ ./my-app/
4  RUN cd my-app && npm install && npm run build
5
6  FROM node:10 AS server-build
7  WORKDIR /root/
8  COPY --from=ui-build /usr/src/app/my-app/out ./my-app/out
9  COPY api/package*.json ./api/
10 RUN cd api && npm install
11 COPY api/server.js ./api/
12
13 EXPOSE 3080
14
15 CMD ["node", "./api/server.js"]
```

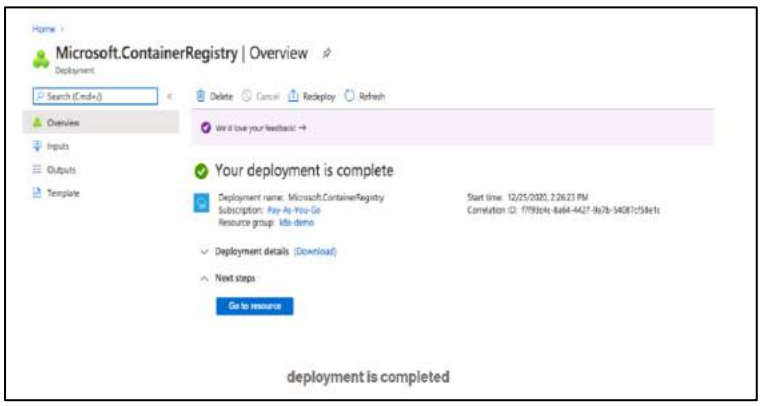
Dockerizing Next.js App With NodeJS Backend

```
// create an image
docker build -t next-node-image .

// running the image
docker run -it -p  3080:3080 --name next-node-ui next-node-image

// list the image you just built
docker images

// list the container
docker ps
```

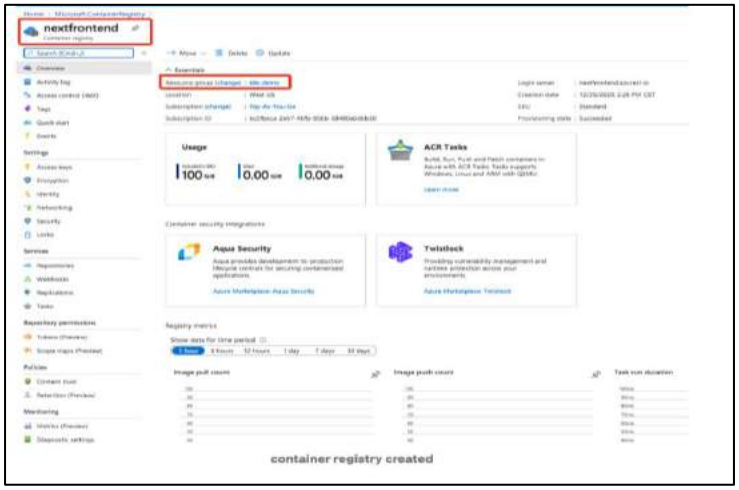
Pushing Docker Image To Container Registry



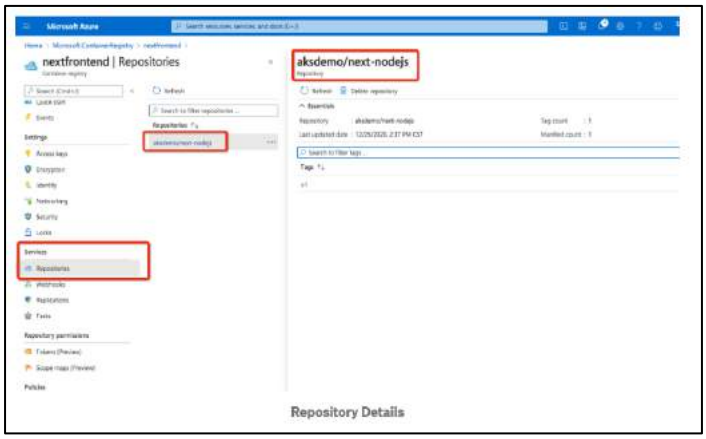
Creating Container Registry



deployment is completed



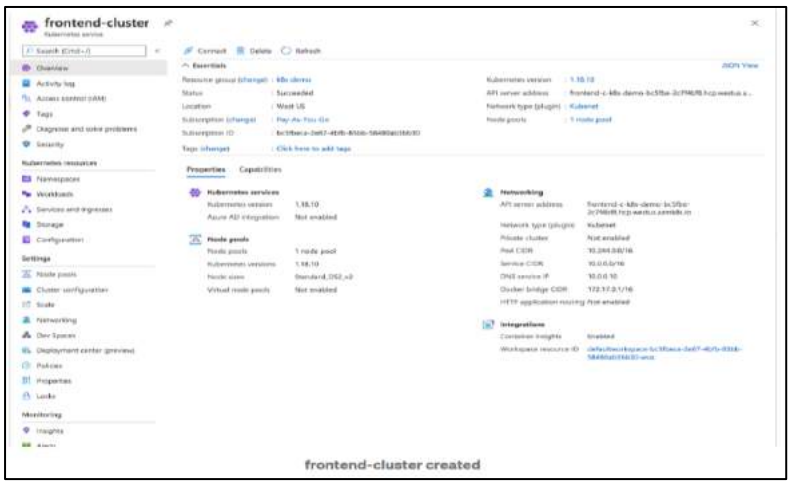
container registry created



Repository Details

docker pull command

Creating AKS Cluster



frontend-cluster created

Configure Kuebctl With AKS Cluster

```
// install CLI
az aks install-cli

// connect to your cluster
az aks get-credentials --resource-group k8s-demo --name frontend-
cluster

// get all the contexts
kubectl config get-contexts

// verify the current context
kubectl config current-context

// get the node
kubectl get nodes
```

```
Bhargavs-MacBook-Pro:java bhargavbachina$ kubectl config current-context
frontend-cluster
Bhargavs-MacBook-Pro:java bhargavbachina$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-nodepool1-95737313-vmss000000   Ready    agent   4m7s    v1.18.10
aks-nodepool1-95737313-vmss000001   Ready    agent   4m25s   v1.18.10
aks-nodepool1-95737313-vmss000002   Ready    agent   4m25s   v1.18.10
Bhargavs-MacBook-Pro:java bhargavbachina$
```

Configure kubectl with AKS Cluster

Deploy Kubernetes Objects on Azure AKS Cluster

```
// list the deployment
kubectl get deploy

// list the pods
kubectl get po

// list the service
kubectl get svc
```

We can see 5 pods running since we have defined 5 replicas for the deployment.

```
Bhargavs-MacBook-Pro:~ bhargavbachina$ kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
next-webapp   5/5     5            5           88s
Bhargavs-MacBook-Pro:~ bhargavbachina$ kubectl get po
NAME                           READY   STATUS    RESTARTS   AGE
next-webapp-866cd685d6-4pv4r   1/1     Running   0          95s
next-webapp-866cd685d6-4z59g   1/1     Running   0          95s
next-webapp-866cd685d6-5cckl   1/1     Running   0          95s
next-webapp-866cd685d6-dtwtq   1/1     Running   0          95s
next-webapp-866cd685d6-lwj2z   1/1     Running   0          95s
Bhargavs-MacBook-Pro:~ bhargavbachina$ kubectl get svc
NAME          TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE
kubernetes    ClusterIP      10.0.0.1       <none>           443/TCP          15m
next-webapp   LoadBalancer   10.0.136.86    20.189.140.128   3080:32468/TCP   109s
```

Kubernetes Objects Running on Azure AKS

Access the WebApp from the browser

```
http://20.189.140.128:3080
```

Accessing from the browser

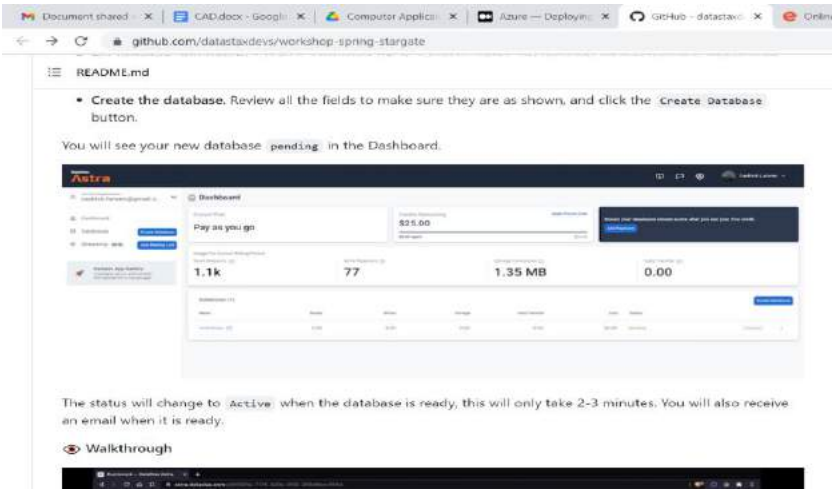
5 Create an AKS cluster

a. from the portal

b. with Azure CLI

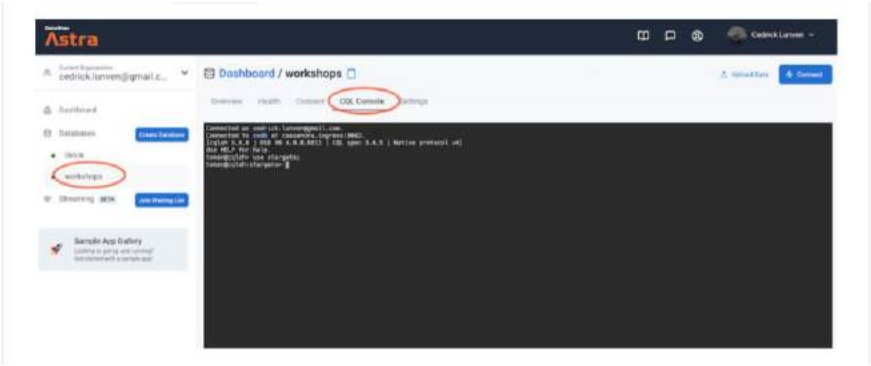
Step 1a. If you do have an account yet register and sign In to Astra DB this is FREE and NO CREDIT CARD

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

```
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 1, 'Earth', 'https://github.com/datas
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 2, 'Crater', 'https://github.com/data
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 3, 'Virgo', 'https://github.com/datas
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 4, 'Bootes', 'https://github.com/data
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 5, 'Centaurus', 'https://github.com/c
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 6, 'Libra', 'https://github.com/datas
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 7, 'Serpenscaput', 'https://github.co
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 8, 'Norma', 'https://github.com/datas
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 9, 'Scorpio', 'https://github.com/dat
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 10, 'Cra', 'https://github.com/datast
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 11, 'Scutum', 'https://github.com/dat
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 12, 'Sagitarus', 'https://github.com/
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 13, 'Aquila', 'https://github.com/dat
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 14, 'Mic', 'https://github.com/datast
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 15, 'Capricorn', 'https://github.com/
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 16, 'Piscesaustrinus', 'https://githu
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 17, 'Equuleus', 'https://github.com/c
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 18, 'Aquarius', 'https://github.com/c
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 19, 'Pegasus', 'https://github.com/da
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 20, 'Sculptor', 'https://github.com/c
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 21, 'Pisces', 'https://github.com/dat
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 22, 'Andromeda', 'https://github.com/
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 23, 'Triangulum', 'https://github.com
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 24, 'Aries', 'https://github.com/data
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 25, 'Perseus', 'https://github.com/da
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 26, 'Cetus', 'https://github.com/data
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 27, 'Taurus', 'https://github.com/dat
INSERT INTO chevrons (area, code, name, picture) VALUES ('Milky Way', 28, 'Auriga', 'https://github.com/dat
```

Step 2e: Show the results

- Validate the number of chevrons

```
select count(*) from stargate.chevrons;
```

- Show the chevrons known for the `Milky Way` galaxy

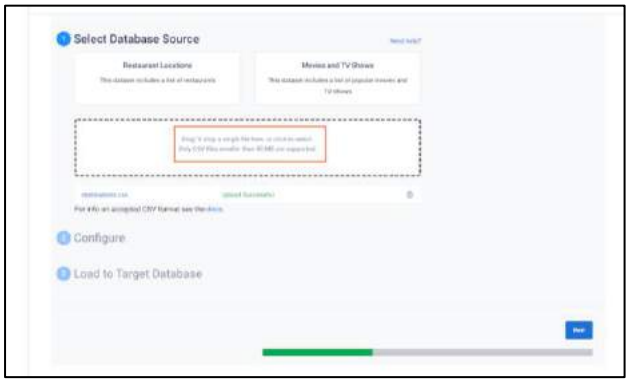
```
select code,name from stargate.chevrons where area='Milky Way';
```

Step 3a: Download the dataset

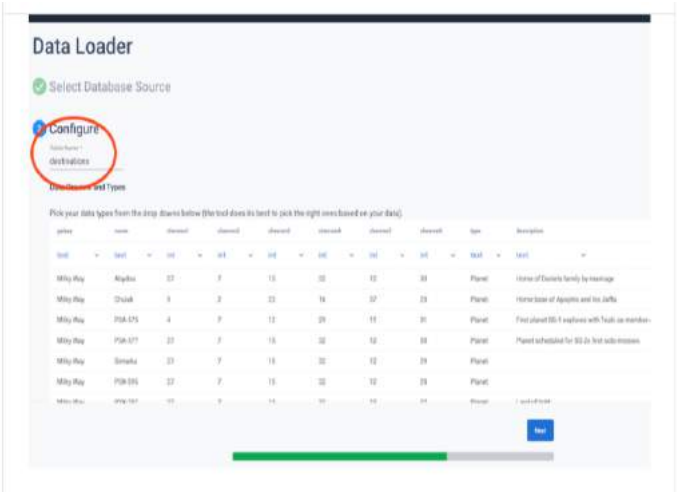
```
galaxy,name,chevron1,chevron2,chevron3,chevron4,chevron5,chevron6,type,description
Milky Way,Abydos,27,7,15,32,12,30,Planet,Home of Daniels family by marriage
Milky Way,Chulak,9,2,23,16,37,20,Planet,Home base of Apophis and his Jaffa
Milky Way,P3A-575,4,7,12,29,11,31,Planet,First planet SG-1 explores with Tealc as member of the unit
Milky Way,P3A-577,27,7,15,32,12,30,Planet,Planet scheduled for SG-2s first solo mission.
Milky Way,Simarka,27,7,15,32,12,29,Planet,
Milky Way,P3X-595,27,7,15,32,12,28,Planet,
Milky Way,P3X-797,27,7,15,32,12,27,Planet,Land of light
Milky Way,P3X-513,27,7,15,32,12,26,Planet,
Milky Way,P3X-562,27,7,15,32,12,28,Planet,
Milky Way,P3X-513,27,7,15,32,12,29,Planet,
Milky Way,P3C-117,27,7,15,32,12,30,Planet,
Milky Way,P3X-774,27,7,15,32,12,31,Planet,
```

Step 3b: Open Astra Data Loader

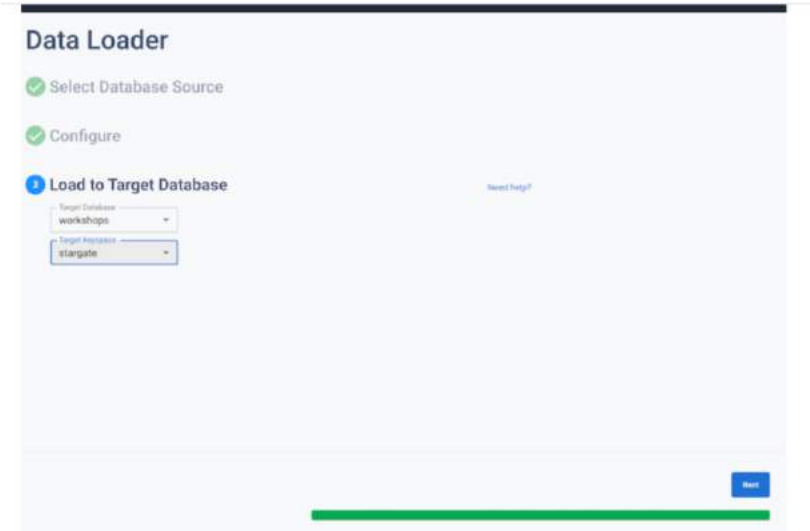
Step 3c: Upload the dataset



Step 3d: Define the target table



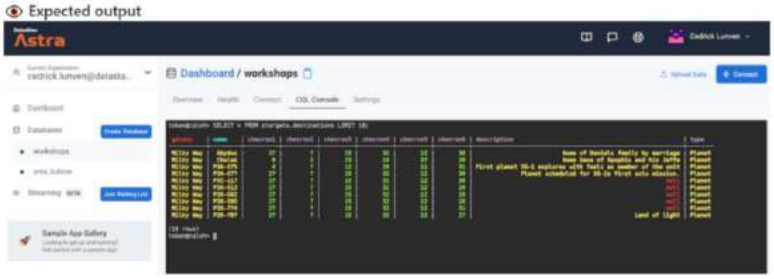
Step 3e: Define the target keyspace



Step 3f: Show Data

```
SELECT * FROM stargate.destinations LIMIT 10;
```

Expected output



6. 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. 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.

PrerequisitesTo 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.

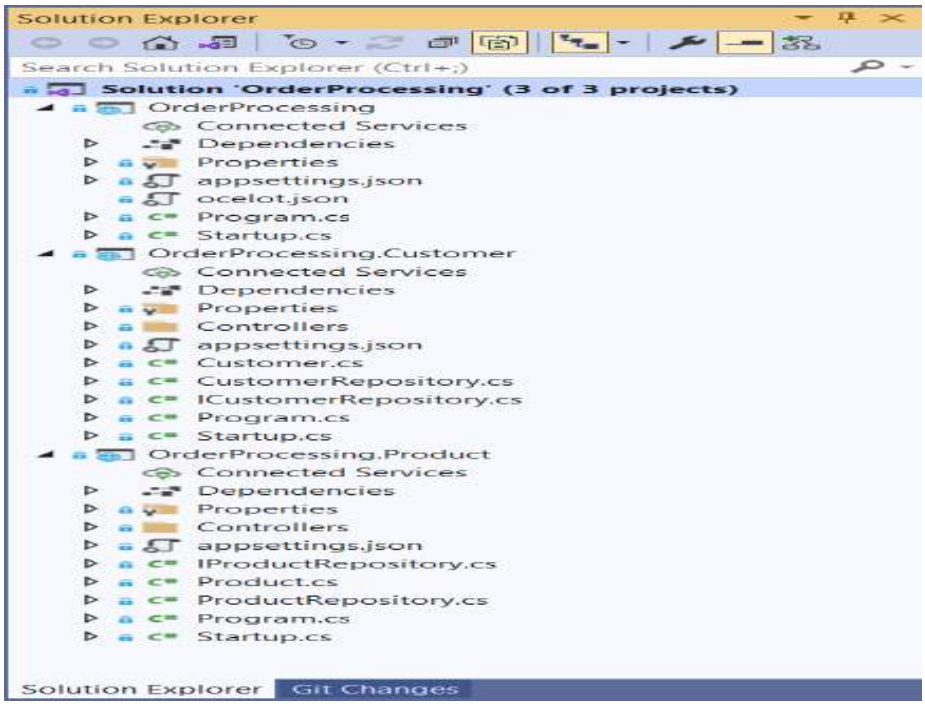
The Customer microservice project will comprise the following classes and interfaces:

- 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.
- ProductRepository class – This is the product repository class that implements the IProductRepository interface.
- 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 = "joydipkanjilal@yahoo.com"
        });

        customers.Add(new Customer()
        {
            Id = Guid.NewGuid(),
            FirstName = "Steve",
            LastName = "Smith",
            EmailAddress = "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:

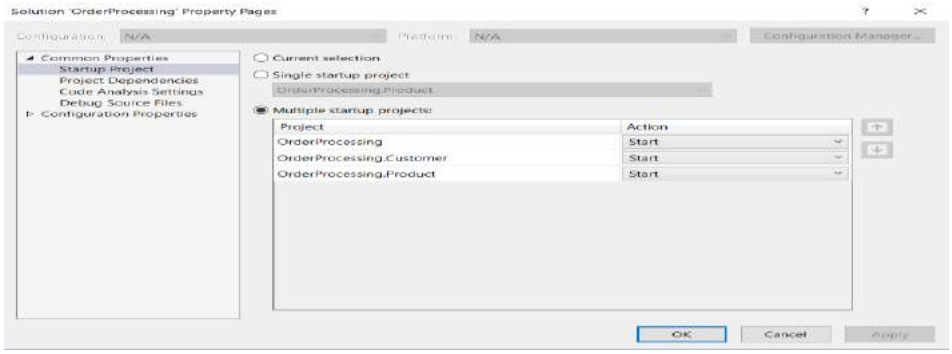
```
// OrderProcessing/Startup.cs

// ... 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:

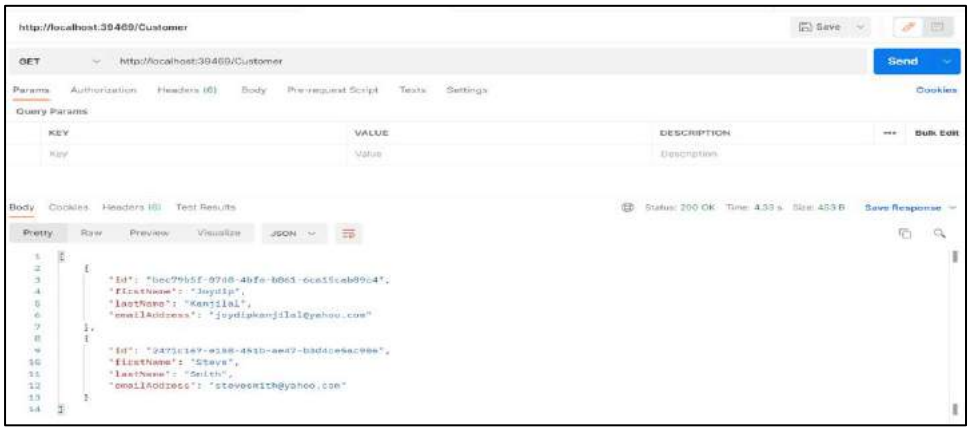


4. Click Ok.

Press the F5 key to run the application. Now send an HTTP Get request to the following URL from Postman or any other HTTP client of your choice:

http://localhost:39469/Customer

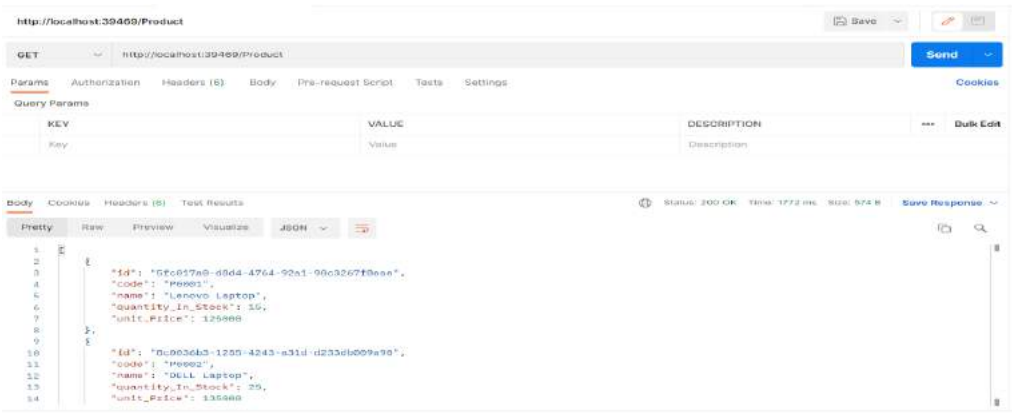
The HTTP Get method of the Customer controller will be executed and the output will look like this:



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 when rate limiting occurs.

- 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

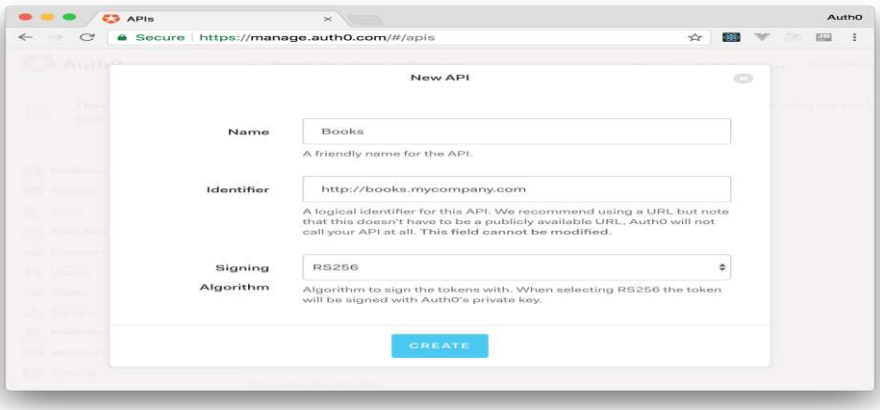
Choosing an exemplary architecture for the needs of your business is the first and foremost step in building applications that are flexible, scalable, and high performant. One of the most significant advantages of microservices architecture is its support for heterogeneous platforms and technologies.

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.

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, you only have to write a few lines of code to get a solid identity management solution, single sign-on, support for social identity providers (like Facebook, GitHub, Twitter, etc.), and support for enterprise identity providers (like Active Directory, LDAP, SAML, custom, etc.).

On ASP.NET Core, you need to create an API in your Auth0 Management Dashboard and change a few things on your code. To create an API, you need to sign up for a free Auth0 account. After that, you need to go to the API section of the dashboard 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", 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();
});
```

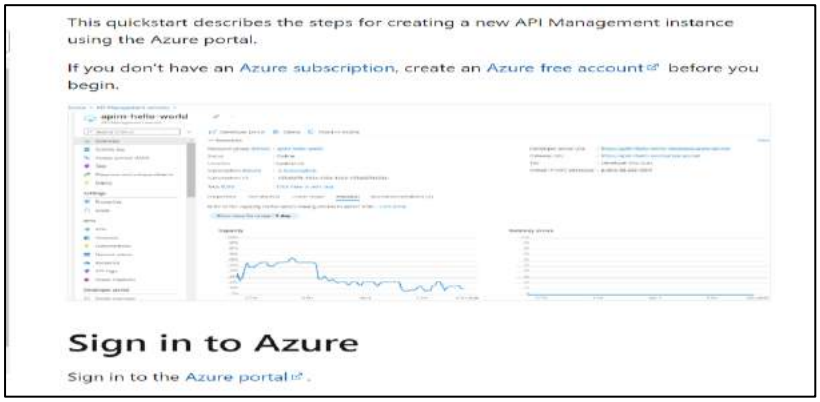
Make sure you invoke these methods in the order shown above. It is essential so that everything works properly.
Finally, add the following element to the appsettings.json configuration file:

```
{
  "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.

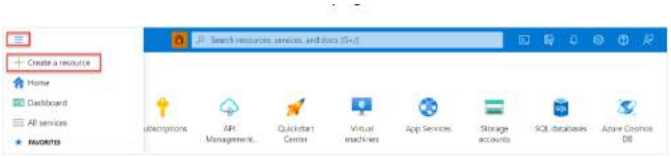
7.Create a database design for Microservices an application using the database.

8 a. Create an API management service

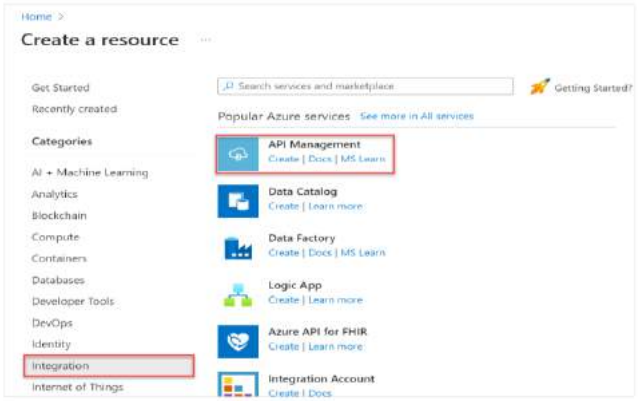


Create a new service

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

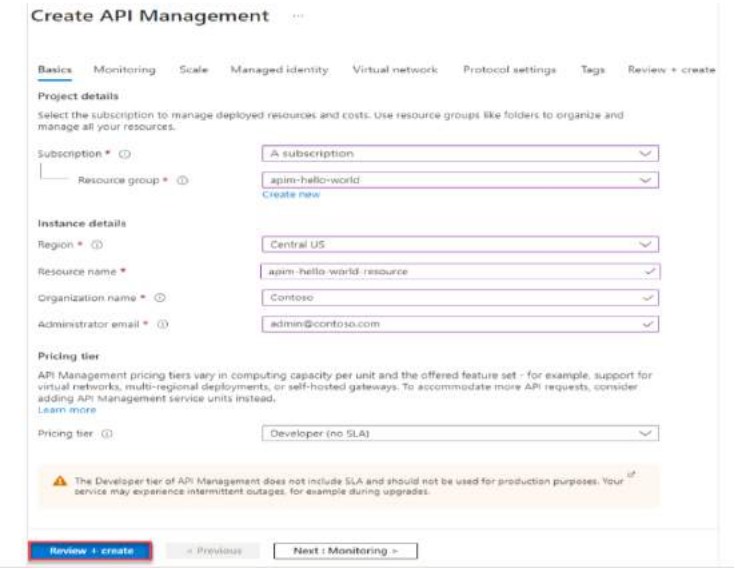


2. On the Create a resource page, select Integration > API Management.



3. In the **Create API Management** page, enter settings.

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

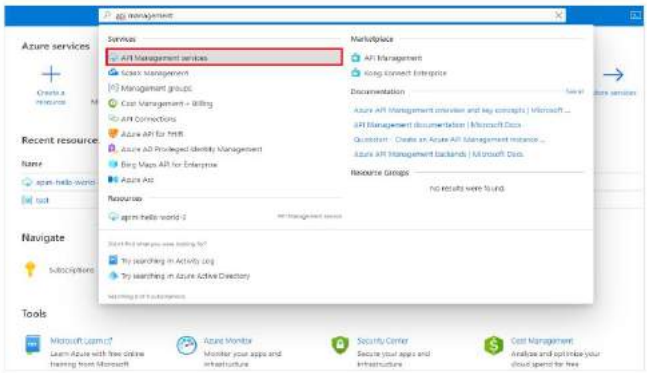


Setting	Description
Subscription	The subscription under which this new service instance will be created.
Resource group	Select a new or existing resource group. A resource group is a logical container into which Azure resources are deployed and managed.
Region	Select a geographic region near you from the available API Management service locations.
Resource name	A unique name for your API Management service. The name can't be changed later. The service name refers to both the service and the corresponding Azure resource. The service name is used to generate a default domain name: *<name>.azure-api.net*. If you would like to configure a custom domain name later, see Configure a custom domain.
Organization name	The name of your organization. This name is used in many places, including the title of the developer portal and sender of notification emails.
Administrator email	The email address to which all the notifications from **API Management** will be sent.
Pricing tier	Select **Developer** tier to evaluate the service. This tier isn't for production use. For more information about scaling the API Management tiers, see upgrade and scale.

4. Select Review + create.

Go to your API Management instance

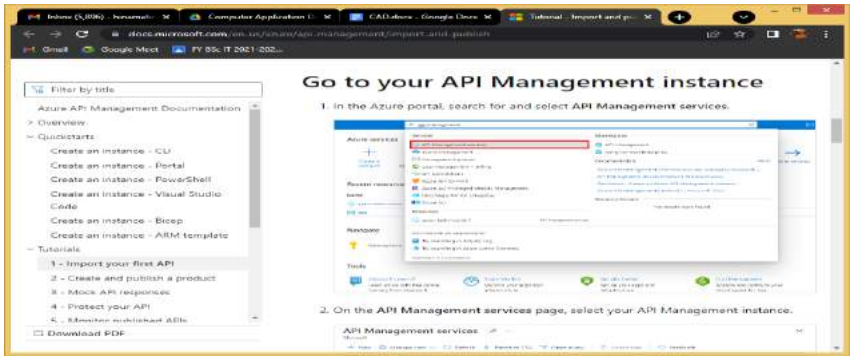
1. In the Azure portal, search for and select API Management services.

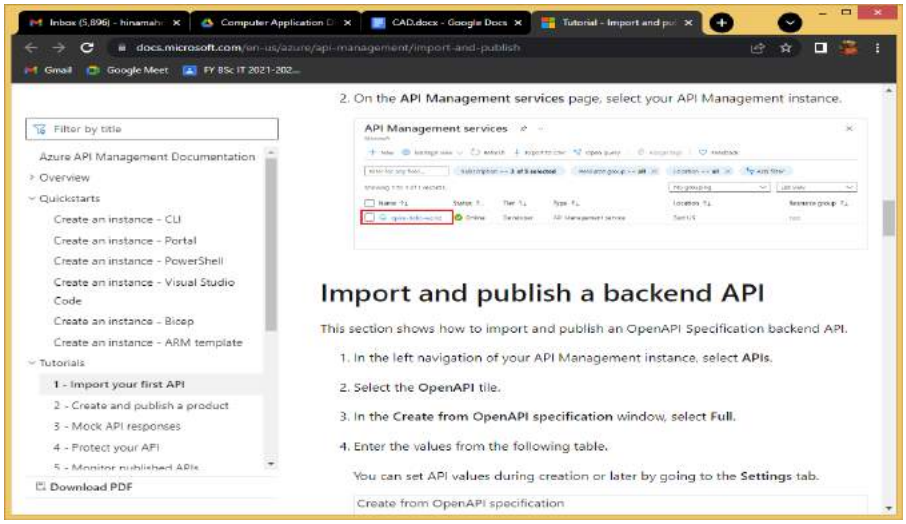


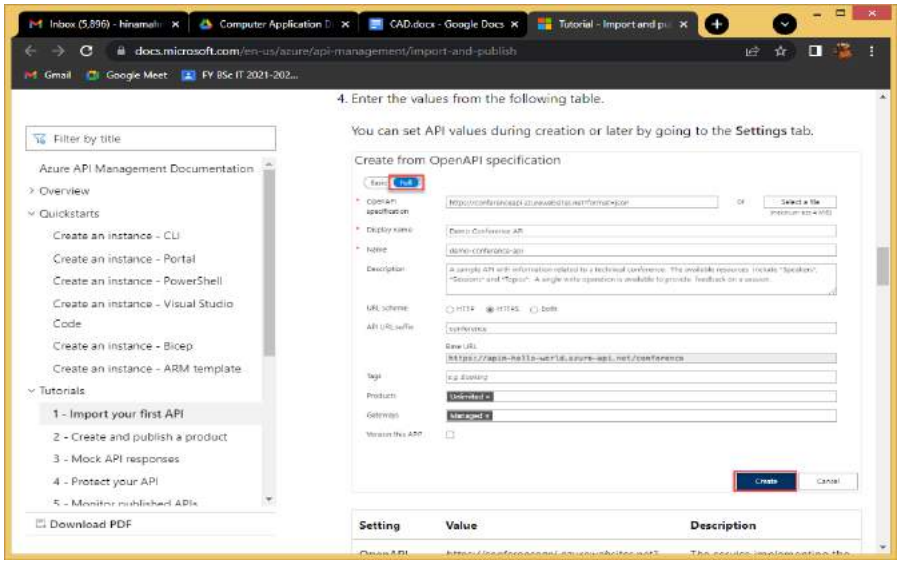
2. On the API Management services page, select your API Management instance.

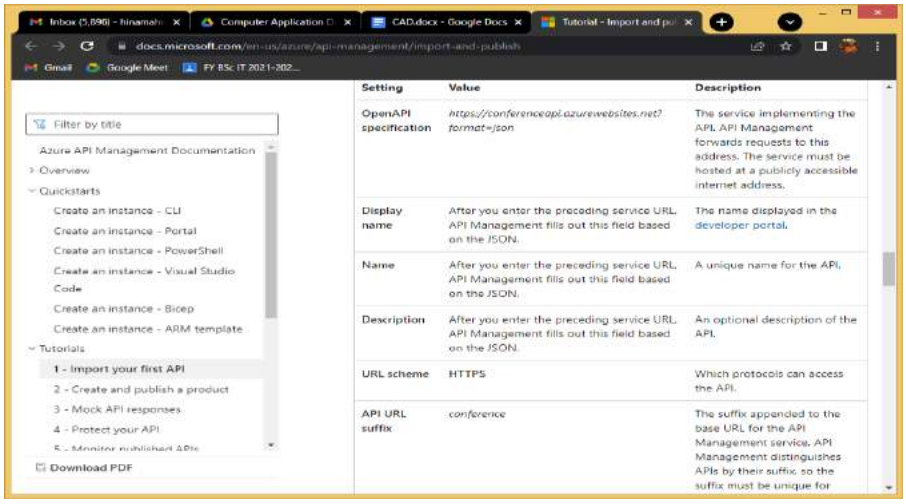
Review the properties of your service on the **Overview** page.

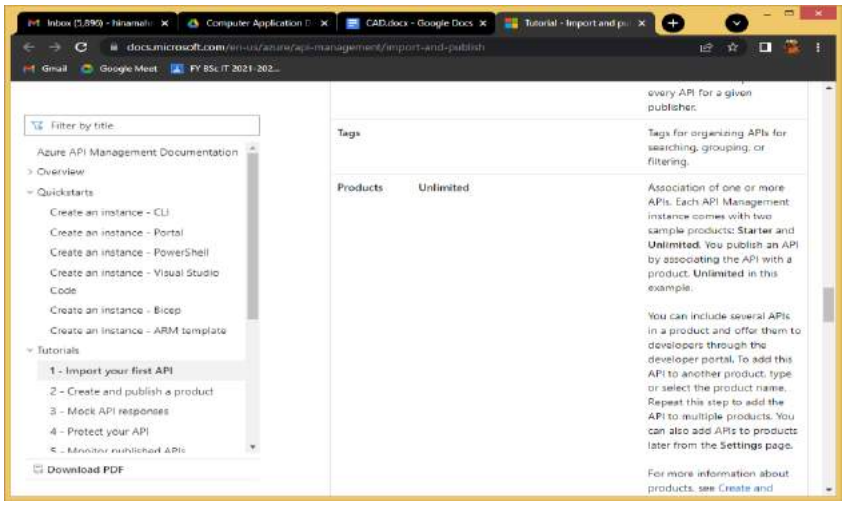
8 b. Create an API gateway service

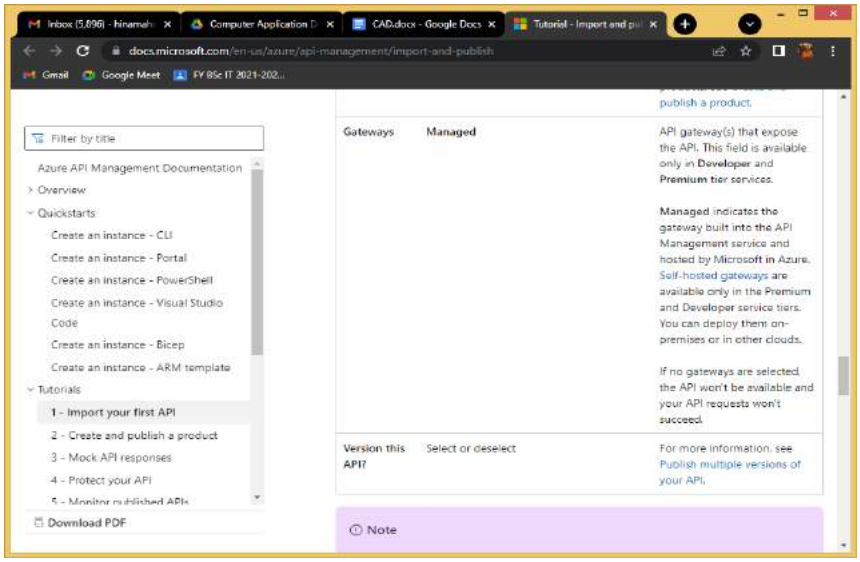


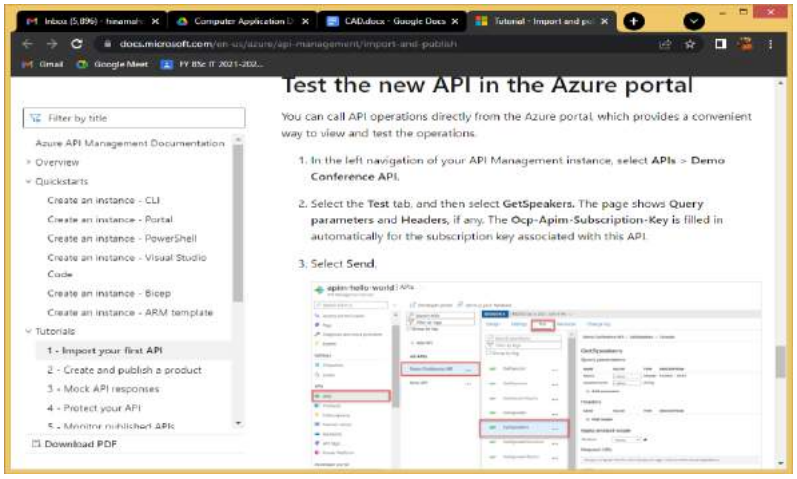


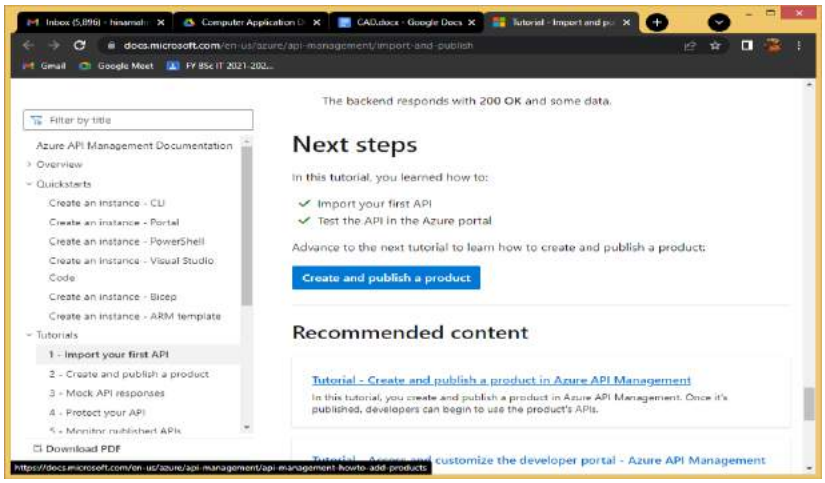



Filter by title
Azure API Management Documentation
Overview
Quickstarts
Create an instance - CLI
Create an instance - Portal
Create an instance - PowerShell
Create an instance - Visual Studio Code
Create an instance - Bicep
Create an instance - ARM template
Tutorials
1 - Import your first API
2 - Create and publish a product
3 - Mock API responses
4 - Protect your API
Download PDF
Setting
Value
Description
OpenAPI specification
https://conferenceapi.azurewebsites.net?format=json
The service implementing the API. API Management forwards requests to this address. The service must be hosted at a publicly accessible internet address.
Display name
After you enter the preceding service URL, API Management fills out this field based on the JSON.
The name displayed in the developer portal.
Name
After you enter the preceding service URL, API Management fills out this field based on the JSON.
A unique name for the API.
Description
After you enter the preceding service URL, API Management fills out this field based on the JSON.
An optional description of the API.
URL scheme
HTTPS
Which protocols can access the API.
API URL suffix
conference
The suffix appended to the base URL for the API Management service. API Management distinguishes APIs by their suffix, so the suffix must be unique for


Filter by title
Azure API Management Documentation
Overview
Quickstarts
Create an instance - CLI
Create an instance - Portal
Create an instance - PowerShell
Create an instance - Visual Studio Code
Create an instance - Bicep
Create an instance - ARM template
Tutorials
1 - Import your first API
2 - Create and publish a product
3 - Mock API responses
4 - Protect your API
Download PDF
every API for a given publisher.
Tags
Tags for organizing APIs for searching, grouping, or filtering.
Products
Unlimited
Association of one or more APIs. Each API Management instance comes with two sample products: Starter and Unlimited. You publish an API by associating the API with a product. Unlimited in this example.
You can include several APIs in a product and offer them to developers through the developer portal. To add this API to another product, type or select the product name. Repeat this step to add the API to multiple products. You can also add APIs to products later from the Settings page.
For more information about products, see Create and


Filter by title
Azure API Management Documentation
Overview
Quickstarts
Create an instance - CLI
Create an instance - Portal
Create an instance - PowerShell
Create an instance - Visual Studio Code
Create an instance - Bicep
Create an instance - ARM template
Tutorials
1 - Import your first API
2 - Create and publish a product
3 - Mock API responses
4 - Protect your API
Download PDF
publish a product.
Gateways
Managed
API gateway(s) that expose the API. This field is available only in Developer and Premium tier services.
Managed indicates the gateway built into the API Management service and hosted by Microsoft in Azure. Self-hosted gateways are available only in the Premium and Developer service tiers. You can deploy them on-premises or in other clouds.
If no gateways are selected, the API won't be available and your API requests won't succeed.
Version this API?
Select or deselect
For more information, see Publish multiple versions of your API.
Note



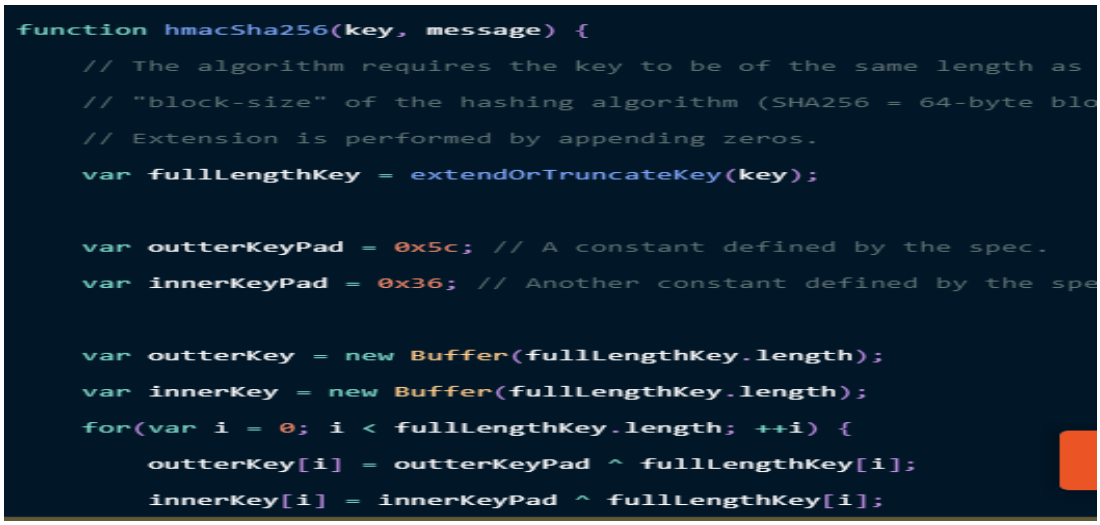


9. Demonstrate

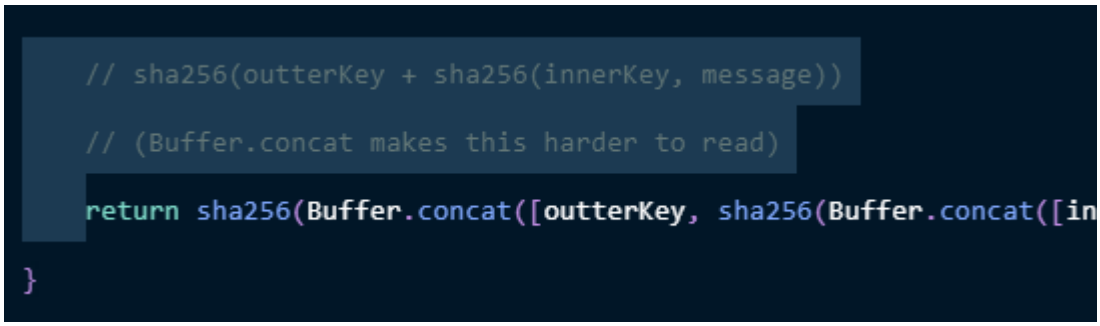
a. Securing APIs with Azure Active Directory.

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.

b. Issuing a custom JWT token using a symmetric signing key



```
function hmacSha256(key, message) {
    // The algorithm requires the key to be of the same length as
    // "block-size" of the hashing algorithm (SHA256 = 64-byte blo
    // Extension is performed by appending zeros.
    var fullLengthKey = extendOrTruncateKey(key);

    var outterKeyPad = 0x5c; // A constant defined by the spec.
    var innerKeyPad = 0x36; // Another constant defined by the spe

    var outterKey = new Buffer(fullLengthKey.length);
    var innerKey = new Buffer(fullLengthKey.length);
    for(var i = 0; i < fullLengthKey.length; ++i) {
        outterKey[i] = outterKeyPad ^ fullLengthKey[i];
        innerKey[i] = innerKeyPad ^ fullLengthKey[i];
```



```
    // sha256(outterKey + sha256(innerKey, message))
    // (Buffer.concat makes this harder to read)
    return sha256(Buffer.concat([outterKey, sha256(Buffer.concat([in
}
```

c. Pre-Authentication in Azure API Management

d. 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.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. 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.

```
{
 "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.

- 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

Type	Example	Notes
Header	$request.header	Header

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

extvariable	text.*variableName*	alueofassupportedcontextvariable.

Stagevariable	$stageVariables.*variableName*	The value of a stage variable.

Caching authorizer responses

You can enable caching for a Lambda authorizer by specifying an 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 def123
```

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 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.

10. Create a serverless API using Azure functions

How to build serverless APIs with Azure Functions

Build serverless APIs

Azure Functions are great for running small pieces of code. They come with many types of triggers and bindings, which take care of connecting input and output data and make development easy. You can host Azure Functions in a consumption plan, which is a serverless plan that scales automatically and that you only pay for when your Function runs.

You can easily create an API with serverless Functions. Each API call is executed by a Function that spins up and scales automatically. By using Functions for your API, you don't have to worry about scaling and you only pay for what you use.

In this post, we'll take a look at how you can create a serverless API with Azure Functions.

Prerequisites

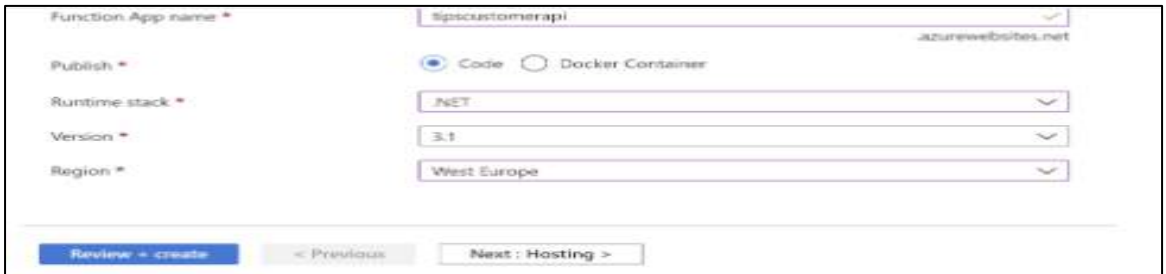
If you want to follow along, you'll need the following:

- An Azure subscription (If you don't have an Azure subscription, create a free account before you begin)

Build a serverless API

Let's create an Azure Function in the Azure portal and turn that into an API.

1. Click the **Create a resource** button (the plus-sign in the top left corner)
2. Search for **Functions**, select the "Function App" result and click **Create**
    1. This brings you to the **Create Function App** blade
    2. Select a **Resource Group**
    3. Fill in a **Name** for the service
    4. Select the **Runtime stack**, which in this demo will be **.NET**
4. Select the **Runtime stack**, which in this demo will be **.NET**
5. Choose the **Region** for the service
6. Click **Next: hosting**
7. Check that the **Plan type** is set to **Consumption (Serverless)**
8. Click **Next: Monitoring**
9. Select **No** for **Enable Application Insights**
10. Click **Review + create** and **Create** after that to create the Function App



(Create an Azure Function in the Azure portal)

When the Function App is created, navigate to it in the Azure portal. Function Apps contain one or more Functions. Let's create a new Function.

1. In the Function App, in the Azure portal, navigate to the **Functions blade**
2. Click **Add** to start adding a Function. This opens the **Add function blade**
    1. Select **Develop in portal** for the Development environment setting
    2. Pick **HTTP trigger** as the **Template**
    3. Give the Function a **Name**
    4. Select **Anonymous** for the **Authorization level**
    5. Click **Add** to create the Function

(Add a Function in the Azure portal)

When the Function is created, it will open in the portal. We'll turn it into an API and try it out.

1. In the Function in the portal, click on the **Integration** menu. This will show the trigger, inputs and outputs for the Function
2. The trigger for the Function is the HTTP trigger. Click on **HTTP (req)**, to edit the trigger
3. By default, you would trigger the Function by navigating to its endpoint and performing an HTTP operation, like a GET or POST against it. That is the behavior of most APIs. However, you might want to change the way you fire HTTP requests at the Function, to make it more of a REST API.
    1. Add a **Route template** to the Function. Type "**customer/{customerName}**". This will expose the HTTP trigger as **functionurl/api/customer**
    2. Click **Save** to save the changes

(Edit the HTTP trigger in the Azure portal)

(Edit the HTTP trigger in the Azure portal)

4. Next, we'll edit the code of the Function. Click on the **Code Test menu**. This opens the code editor
5. The Function code comes from the standard HTTP trigger template. You need to change it into the code below, which contains two changes:
    1. The **Run** method now contains a **string customerName** which catches the customerName parameter of the API route
    2. **string name** now comes from the **customerName parameter**

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static async Task<IActionResult> Run(HttpRequ
{
    log.LogInformation("C# HTTP trigger function pro

    string name = customerName;

    string requestBody = await new StreamReader(req.
    dynamic data = JsonConvert.DeserializeObject(req
    name = name ?? data?.name;

    string responseMessage = string.IsNullOrEmpty(na
        ? "This HTTP triggered function executed suc
        : $"Hello, {name}. You are a custome

            return new OkObjectResult(responseMessag
}
```

6. Click **Save** to save the changes and compile the code
7. Select **Test/Run** to open the test window
8. Fill in a **value** for the **customerName Query parameter**
9. Next, click **Run** to try out the code
10. If the call succeeded, you should see the value that you entered in the **Output** tab

(Code and Test in the Azure portal)

11. Let's see if we can call the Function like an API. Click on the **Get Function URL** button and **copy** the URL
12. Paste the URL in a new browser window and replace the **{customerName}** at the end of the URL with a value. The URL would look something like https://functionappname.azurewebsites.net/api/customer/mynewcus
13. When you submit the URL, you should see the value of the customer parameter in the result

(Calling a Function like an API)

We have just created an API endpoint using a serverless Azure Function. To create a complete API, you can create multiple Functions that each have their own purpose and are exposed through other URL endpoints.

Conclusion

Azure Functions are extremely well suited to run APIs at scale, because they can be triggered by HTTP requests, and because they scale automatically. Also, when you run them serverless, you only pay for them when they run. Go and check it out!

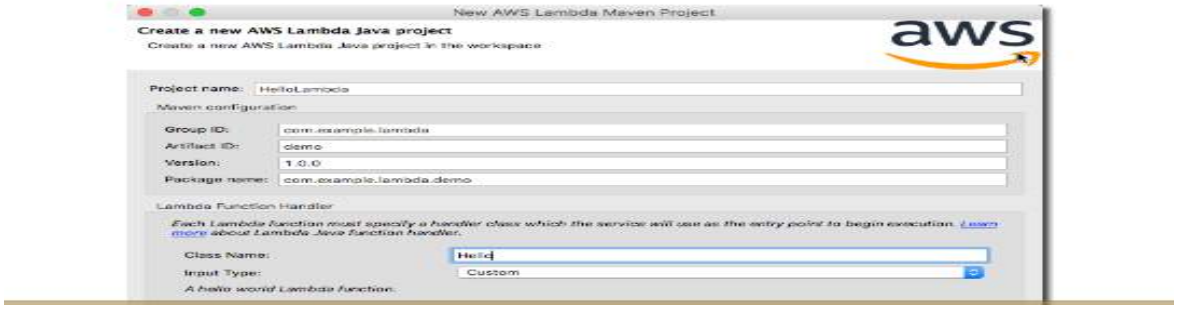
11. Create an AWS Lambda function

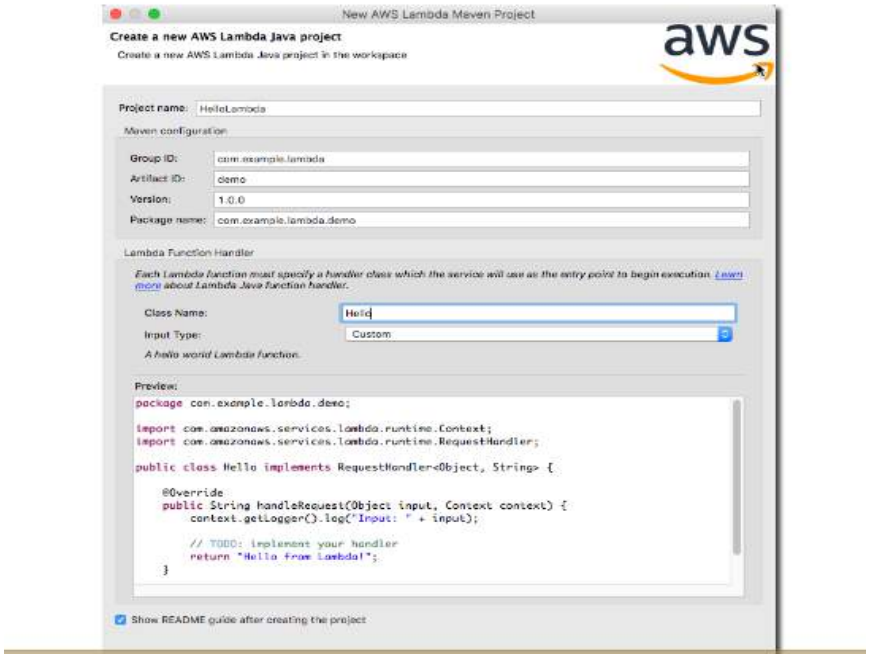
To create an AWS Lambda project

1. On the Eclipse toolbar, open the Amazon Web Services menu (identified by the AWS homepage icon), and then choose **New AWS Lambda Java project**. Or on the Eclipse menu bar, choose **File**, **New**, **AWS Lambda Java Project**.
2. Add a *Project name*, *Group ID*, *Artifact ID*, and *class name* in the associated input boxes. The Group ID and Artifact ID are the IDs that identify a Maven build artifact. This tutorial uses the following example values:
   - **Project name**: *HelloLambda*
   - **Group ID**: *com.example.lambda*
   - **Artifact ID**: *demo*
   - **Class name**: *Hello*

   The **Package Name** field is the package namespace for the AWS Lambda handler class. The default value for this field is a concatination of the Group ID and Artifact ID, following Maven project conventions. This field is automatically updated when the **Group ID** and **Artifact ID** fields are updated.

3. For **Input Type**, choose **Custom**. For information about each of the available input types, see New AWS Lambda Java Project Dialog.
4. Verify that your entries look like the following screenshot (modify them if they are not), and then choose **Finish**.





As you type, the code in the **Source preview** changes to reflect the changes you make in the dialog box.

5. After you choose **Finish**, your project's directory and source files are generated in your Eclipse workspace. A new web browser window opens, displaying `README.html` (which was created for you in your project's root directory). `README.html` provides instructions to guide you through the next steps of implementing, testing, uploading, and invoking your new Lambda function. Read through it to gain some familiarity with the steps that are described here.

Next, you implement the function in the `HelloLambda` Java project that was just created for you in Eclipse.

Implement the Handler Method

You use the **Create New Project** dialog box to create a skeleton project. Now fill in the code that will be run when your Lambda function is invoked. (In this case, by a custom event that sends a String to your function, as you specified when setting your method's input parameter.)

To implement your Lambda handler method

1. In the Eclipse **Project Explorer**, open `Hello.java` in the **HelloLambda** project. It will contain code similar to the following.

```
package com.example.lambda.demo;

import com.amazonaws.services.lambda.runtime.Contex
import com.amazonaws.services.lambda.runtime.Reques

public class Hello implements RequestHandler<Object

    @Override
    public String handleRequest(Object input, Conte
        context.getLogger().log("Input: " + input)

        // TODO: implement your handler
        return "Hello from Lambda";
    }

}
```

2. Replace the contents of the `handleRequest` function with the following code.

```
@Override
public String handleRequest(String input, Context c
  context.getLogger().log("Input: " + input);
  String output = "Hello, " + input + "!";
  return output;
}
```

Allow Lambda to Assume an IAM Role

For Lambda to be able to access your Lambda function, you have to create an IAM role that gives it access to your AWS resources. You can create the role in two ways, either through the AWS Management Console or by using the Toolkit for Eclipse. This section describes how to create the IAM role in the console. See Upload the Code to create one using the Toolkit for Eclipse.

To create an IAM role for Lambda

1. Sign in to the AWS Management Console.
2. From the **Services** menu, open the IAM console.
3. In the Navigation pane, choose **Roles**, and then choose **Create role**.
4. For **Select type of trusted entity**, choose **AWS service**, and then choose **Lambda** for the service that will use this role. Then choose **Next: Permissions**.
5. For **Attach permissions policy**, choose **AWSLambdaBasicExecutionRole**. This allows Lambda to write to your CloudWatch Logs resources. Then choose **Next: Review**.
6. Add a name for your role, such as `hello-lambda-role`, and a description for the role. Then choose **Create role** to finish creating the IAM role.

Create an Amazon S3 Bucket for Your Lambda Code

AWS Lambda requires an Amazon S3 bucket to store your Java project when you upload it. You can either use a bucket that already exists in the AWS Region in which you'll run your code, or you can create a new one specifically for Lambda to use (recommended).

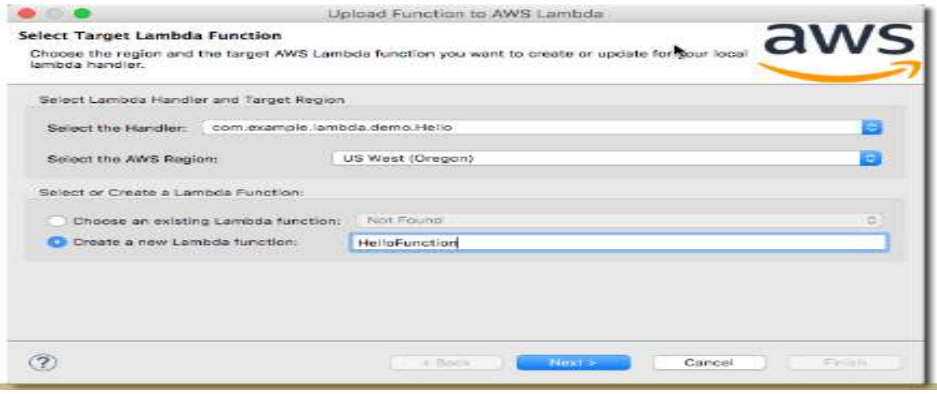
You can create an Amazon S3 bucket in two ways, either through the AWS Management Console or by using the Toolkit for Eclipse. This section describes how to create an Amazon S3 bucket in the

To create an Amazon S3 bucket for use with Lambda

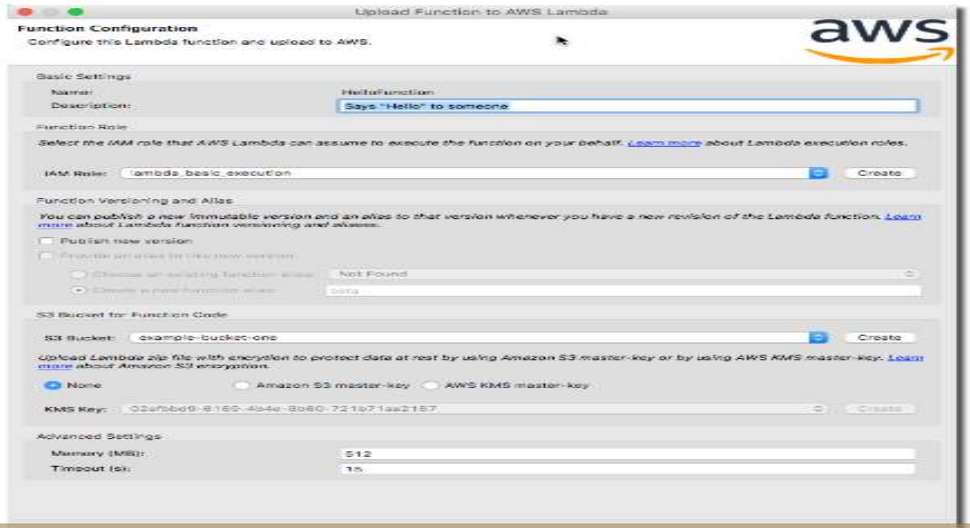
1. Sign in to the AWS Management Console.
2. From the **Services** menu, open the S3 console.
3. Choose **Create bucket**.
4. Enter a bucket name, and then choose a region for your bucket. This region should be the same one in which you intend to run your Lambda function. For a list of regions supported by Lambda see Regions and Endpoints in the *AWS General Reference*.
5. Choose **Create** to finish creating your bucket.

To upload your function to Lambda

1. Right-click in your Eclipse code window, choose **AWS Lambda**, and then choose **Upload function to AWS Lambda**.
2. On the **Select Target Lambda Function** page, choose the AWS Region to use. This should be the same region that you chose for your Amazon S3 bucket.

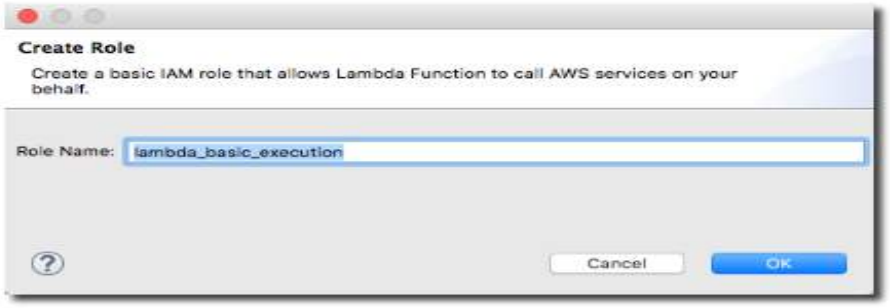


3. Choose **Create a new Lambda function**, and then type a name for your function (for example, `HelloFunction`).
4. Choose **Next**.
5. On the **Function Configuration** page, enter a description for your target Lambda function, and then choose the IAM role and Amazon S3 bucket that your function will use.



For more information about the available options, see Upload Function to AWS Lambda Dialog Box.

6. On the **Function Configuration** page, choose **Create** in **Function Role** if you want to create a new IAM role for your Lambda function. Enter a role name in the dialogue box the **Create Role** dialogue box.



7. On the **Function Configuration** page, choose **Publish new version** if you want the upload to create a new version of the Lambda function. To learn more about versioning and aliases in Lambda, see AWS Lambda Function Versioning and Aliases in the *AWS Lambda Developer Guide*.
8. If you chose to publish a new version, the **Provide an alias to this new version** option is enabled. Choose this option if you want to associate an alias with this version of the Lambda function.
9. On the **Function Configuration** page, choose **Create** in the **S3 Bucket for Function Code** section if you want to create a new Amazon S3 bucket for your Lambda function. Enter a bucket name in the **Create Bucket** dialogue box.

9. On the **Function Configuration** page, choose **Create** in the **S3 Bucket for Function Code** section if you want to create a new Amazon S3 bucket for your Lambda function. Enter a bucket name in the **Create Bucket** dialogue box.

10. In the **S3 Bucket for Function Code** section, you can also choose to encrypt the uploaded code. For this example, leave **None** selected. To learn more about Amazon S3 encryption, see Protecting Data Using Server-Side Encryption in the *Amazon Simple Storage Service User Guide*.

11. Leave the **Advanced Settings** options as they are. The Toolkit for Eclipse selects default values for you. Choose **Finish** to upload your Lambda function to AWS.

If the upload succeeds, you will see the Lambda function name that you chose appear next to your Java handler class in the **Project Explorer** view.

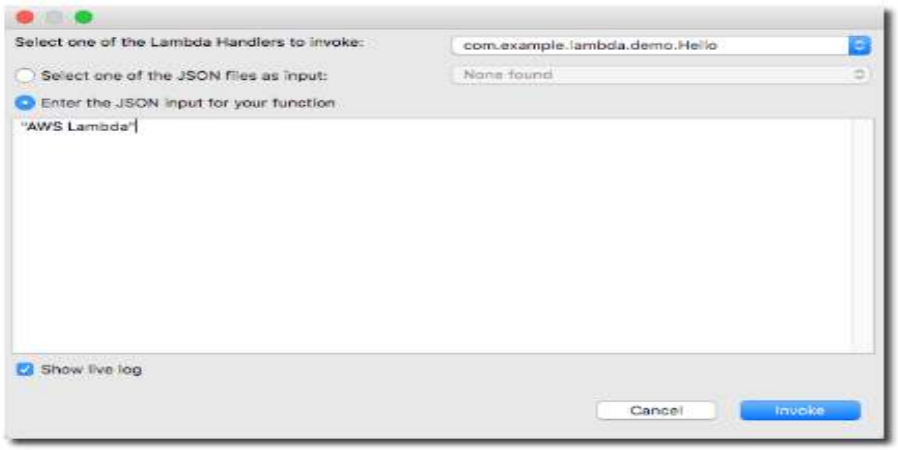
If you don't see this happen, open the Eclipse **Error Log** view. Lambda writes information about failures to upload or run your function to this error log so you can debug them.

Invoke the Lambda Function

You can now invoke the function on AWS Lambda.

To invoke your Lambda function

1. Right-click in the Eclipse code window, choose **AWS Lambda**, and then choose **Run Function on AWS Lambda**.
2. Choose the handler class you want to invoke.
3. In the input box, type a valid JSON string, such as "AWS Lambda".



> **Note**
>
> You can add new JSON input files to your project, and they will show up in this dialog box if the file name ends with .json. You can use this feature to provide standard input files for your Lambda functions.

4. The **Show Live Log** box is checked by default. This displays the logs from the Lambda function output in the Eclipse **Console**.
5. Choose **Invoke** to send your input data to your Lambda function. If you have set up everything correctly, you should see the return value of your function printed out in the Eclipse **Console** view (which automatically appears if it isn't already shown).

Congratulations, you've just run your first Lambda function directly from the Eclipse IDE!

Next Steps

Now that you've uploaded and deployed your function, try changing the code and rerunning the function. Lambda automatically reuploads and invokes the function for you, and prints output to the Eclipse **Console**.

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

12. 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.
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.

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

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

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.
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.

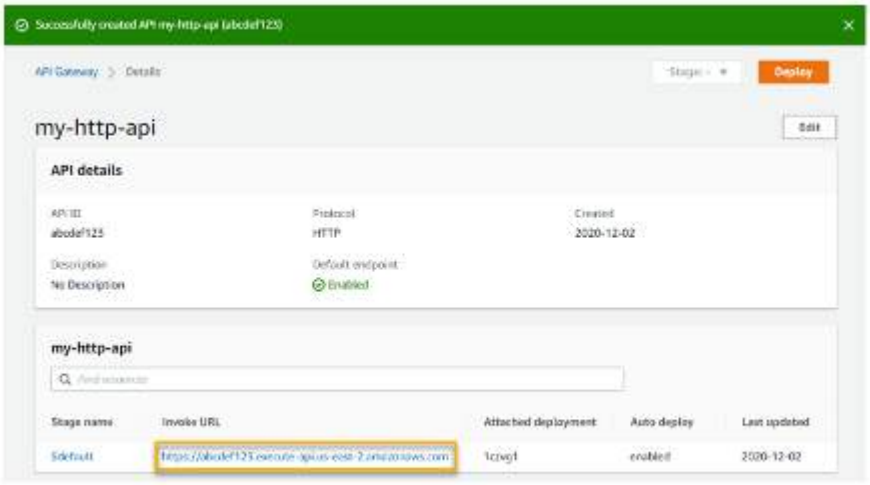
Now you've created an HTTP API with a Lambda integration that's ready to receive requests from clients.

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.
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.
2. 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.
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.
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.
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.
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.