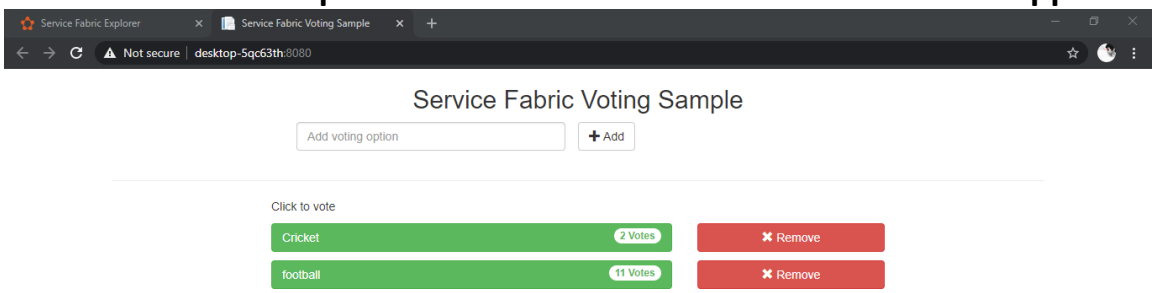
PRACTICAL 01
Develop an ASP.NET Core MVC based Stateful Web App



Step 1: Create Project

Launch Visual Studio as an administrator. Create a project with File >New >Project.

In the New Project dialog, choose Cloud > Service Fabric Application. Name the application Voting and click OK.

On the New Service Fabric Service page, choose Stateless ASP.NET Core, name your service VotingWeb, then click OK.

The next page provides a set of ASP.NET Core project templates. For this practical, choose Web Application (Model-View-Controller), then click OK.

Visual Studio creates an application and a service project and displays them in Solution Explorer.

Step 2: Create and update the files

In this step we will create and will update file in the VotingWeb project

2.1 Update the site.js file

Open wwwroot/js/site.js. Replace its contents with the following JavaScript used by the Home views, then save your changes.

```
var app = angular.module('VotingApp', ['ui.bootstrap']); app.run(function () { });
app.controller('VotingAppController', ['$rootScope', '$scope', '$http',
'$timeout', function ($rootScope,
$scope, $http, $timeout) {
$scope.refresh = function () {
$http.get('api/Votes?c=' + new Date().getTime())
.then(function (data, status) {
$scope.votes = data;
}, function (data, status) {
$scope.votes = undefined;
});
};
$scope.remove = function (item) {
$http.delete('api/Votes/' + item)
.then(function (data, status) {
$scope.refresh();
```

```
})
};
$scope.add = function (item) { var fd = new FormData(); fd.append('item',
item);
$http.put('api/Votes/' + item, fd, { transformRequest: angular.identity,
headers: { 'Content-Type': undefined }
})
.then(function (data, status) {
$scope.refresh();
$scope.item = undefined;
})
};
}]);
```

2.2 Update the Index.cshtml file

Open Views/Home/Index.cshtml, the view specific to the Home controller. Replace its contents with the following, then save your changes.

```
@{
ViewData["Title"] = "Service Fabric Voting Sample";
}
<div ng-controller="VotingAppController" ng-init="refresh()">
<div class="container-fluid">
<div class="row">
<div class="col-xs-8 col-xs-offset-2 text-center">
<h2>Service Fabric Voting Sample</h2>
</div>
</div>
<div class="row">
<div class="col-xs-8 col-xs-offset-2">
<form class="col-xs-12 center-block">
<div class="col-xs-6 form-group">
<input id="txtAdd" type="text" class="form-control" placeholder="Add voting
option" ng- model="item"/>
</div>
<button id="btnAdd" class="btn btn-default" ng-click="add(item)">
<span class="glyphicon glyphicon-plus" aria-hidden="true"></span> Add
</button>
</form>
</div>
</div>
```

```
<hr/>
<div class="row">
<div class="col-xs-8 col-xs-offset-2">
<div class="row">
<div class="col-xs-4"> Click to vote
</div>
</div>
<div class="row top-buffer" ng-repeat="vote in votes.data">
<div class="col-xs-8">
<button class="btn btn-success text-left btn-block" ng-click="add(vote.key)">
<span class="pull-left">
{{vote.key}}
</span>
<span class="badge pull-right">
{{vote.value}} Votes
</span>
</button>
</div>
<div class="col-xs-4">
<button class="btn btn-danger pull-right btn-block" ng-
click="remove(vote.key)">
<span class="glyphicon glyphicon-remove" aria-hidden="true"></span>
Remove
</button>
</div>
</div>
</div>
</div>
</div>
</div>
```

2.3 Update the _Layout.cshtml file

Open Views/Shared/_Layout.cshtml, the default layout for the ASP.NET app. Replace its contents with the following, then save your changes.

```
<!DOCTYPE html>
<html ng-app="VotingApp" xmlns:ng="http://angularjs.org">
<head>
<meta charset="utf-8"/>
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<title>@ViewData["Title"]</title>
```

```
<link href="~/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet"/>
<link href="~/css/site.css" rel="stylesheet"/>
</head>
<body>
<div class="container body-content"> @RenderBody()
</div>
<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
<script src="~/lib/angular/angular.js"></script>
<script src="~/lib/angular-bootstrap/ui-bootstrap-tpls.js"></script>
<script src="~/js/site.js"></script>
@RenderSection("Scripts", required: false)
</body>
</html>
```

2.4 Update the VotingWeb.cs file

Open the VotingWeb.cs file, which creates the ASP.NET Core WebHost inside the stateless service using the WebListener web server.

Replace the content with the following code, then save your changes.

```
namespace VotingWeb
{
using System;
using System.Collections.Generic; using System.Fabric;
using System.IO; using System.Net.Http;
using Microsoft.AspNetCore.Hosting; using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.ServiceFabric.Services.Communication.AspNetCore;
using Microsoft.ServiceFabric.Services.Communication.Runtime; using
Microsoft.ServiceFabric.Services.Runtime;
internal sealed class VotingWeb : StatelessService
{
public VotingWeb(StatelessServiceContext context)
: base(context)
{
}
protected override IEnumerable<ServiceInstanceListener>
CreateServiceInstanceListeners()
{
return new ServiceInstanceListener[]
{
new ServiceInstanceListener( serviceContext =>
```

```
new KestrelCommunicationListener( serviceContext, "ServiceEndpoint",
(url, listener) =>
{
ServiceEventSource.Current.ServiceMessage(serviceContext, $"Starting Kestrel
on {url}");
return new WebHostBuilder()
.UseKestrel()
.ConfigureServices( services => services
.AddSingleton<HttpClient>(new HttpClient())
.AddSingleton<FabricClient>(new FabricClient())
.AddSingleton<StatelessServiceContext>(serviceContext))
.UseContentRoot(Directory.GetCurrentDirectory())
.UseStartup<Startup>()
.UseServiceFabricIntegration(listener, ServiceFabricIntegrationOptions.None)
.UseUrls(url)
.Build();
}))
};
}
internal static Uri GetVotingDataServiceName(ServiceContext context)
{
return new
Uri($"{context.CodePackageActivationContext.ApplicationName}/VotingData")
;
}
}
}
```

2.5 Add the VotesController.cs file

a. Add a controller, which defines voting actions. Right-click on the Controllers folder, then select Add->New item->Visual C#->ASP.NET Core->Class. Name the file VotesController.cs, then click Add.
b. Replace the VotesController.cs file contents with the following, then save your changes this file is modified to read and write voting data from the back-end service.

```
namespace VotingWeb.Controllers
{
using System;
using System.Collections.Generic; using System.Fabric;
using System.Fabric.Query; using System.Linq;
using System.Net.Http;
```

```
using System.Net.Http.Headers; using System.Text;
using System.Threading.Tasks; using Microsoft.AspNetCore.Mvc; using
Newtonsoft.Json;
[Produces("application/json")] [Route("api/[controller]")]
public class VotesController : Controller
{
private readonly HttpClient httpClient; private readonly FabricClient
fabricClient; private readonly string reverseProxyBaseUri;
private readonly StatelessServiceContext serviceContext;
public VotesController(HttpClient httpClient, StatelessServiceContext context,
FabricClient fabricClient)
{
this.fabricClient = fabricClient; this.httpClient = httpClient; this.serviceContext
= context;
this.reverseProxyBaseUri =
Environment.GetEnvironmentVariable("ReverseProxyBaseUri");
}
// GET: api/Votes [HttpGet("")]
public async Task<IActionResult> Get()
{
Uri serviceName =
VotingWeb.GetVotingDataServiceName(this.serviceContext); Uri proxyAddress
= this.GetProxyAddress(serviceName);
ServicePartitionList partitions = await
this.fabricClient.QueryManager.GetPartitionListAsync(serviceName);
List<KeyValuePair<string, int>> result = new List<KeyValuePair<string, int>>();
foreach (Partition partition in partitions)
{
string proxyUrl =
$"{proxyAddress}/api/VoteData?PartitionKey={((Int64RangePartitionInformati
on) partition.PartitionInformation).LowKey}&PartitionKind=Int64Range";
using (HttpResponseMessage response = await
this.httpClient.GetAsync(proxyUrl))
{
if (response.StatusCode != System.Net.HttpStatusCode.OK)
{
continue;
}
result.AddRange(JsonConvert.DeserializeObject<List<KeyValuePair<string,
int>>>(await response.Content.ReadAsStringAsync()));
```

```
}
}
return this.Json(result);
}
// PUT: api/Votes/name [HttpPut("{name}")]
public async Task<IActionResult> Put(string name)
{
Uri serviceName =
VotingWeb.GetVotingDataServiceName(this.serviceContext); Uri proxyAddress
= this.GetProxyAddress(serviceName);
long partitionKey = this.GetPartitionKey(name); string proxyUrl =
$"{proxyAddress}/api/VoteData/{name}?PartitionKey={partitionKey}&Partition
Kind=Int64Range";
StringContent putContent = new StringContent($"{{ 'name' : '{name}' }}",
Encoding.UTF8, "application/json");
putContent.Headers.ContentType = new
MediaTypeHeaderValue("application/json");
using (HttpResponseMessage response = await
this.httpClient.PutAsync(proxyUrl, putContent))
{
return new ContentResult()
{
StatusCode = (int) response.StatusCode,
Content = await response.Content.ReadAsStringAsync()
};
}
}
// DELETE: api/Votes/name [HttpDelete("{name}")]
public async Task<IActionResult> Delete(string name)
{
Uri serviceName =
VotingWeb.GetVotingDataServiceName(this.serviceContext); Uri proxyAddress
= this.GetProxyAddress(serviceName);
long partitionKey = this.GetPartitionKey(name); string proxyUrl =
$"{proxyAddress}/api/VoteData/{name}?PartitionKey={partitionKey}&Partition
Kind=Int64Range";
using (HttpResponseMessage response = await
this.httpClient.DeleteAsync(proxyUrl))
{
if (response.StatusCode != System.Net.HttpStatusCode.OK)
```

```
{
return this.StatusCode((int) response.StatusCode);
}
}
return new OkResult();
}
/// <summary>
/// Constructs a reverse proxy URL for a given service.
/// Example: http://localhost:19081/VotingApplication/VotingData/
/// </summary>
/// <param name="serviceName"></param>
/// <returns></returns>
private Uri GetProxyAddress(Uri serviceName)
{
return new Uri($"{this.reverseProxyBaseUri}{serviceName.AbsolutePath}");
}

/// <summary>
/// Creates a partition key from the given name.
/// Uses the zero-based numeric position in the alphabet of the first letter of
the name (0-25).
/// </summary>
/// <param name="name"></param>
/// <returns></returns>
private long GetPartitionKey(string name)
{
return Char.ToUpper(name.First()) - 'A';
}
}
}
```

Step 3: Add a stateful back-end service to your application

a. In Solution Explorer, right-click Services within the Voting application project and choose Add -> New Service Fabric Service....
b. In the New Service Fabric Service dialog, choose Stateful ASP.NET Core, name the service VotingData, then press Create.

Once your service project is created, you have two services in your application. As you continue to build your application, you can add more services in the same way. Each can be independently versioned and upgraded. The next page provides a set of ASP.NET Core project templates. choose API. Visual Studio creates the VotingData service project and displays it in Solution Explorer.

3.1 Add the VoteDataController.cs file

a. In the VotingData project, right-click on the Controllers folder, then select Add->New item->Class. Name the file VoteDataController.cs and click Add.

b. Replace the file contents with the following, then save your changes.

```
namespace VotingData.Controllers
{
using System.Collections.Generic; using System.Threading;
using System.Threading.Tasks; using Microsoft.AspNetCore.Mvc; using
Microsoft.ServiceFabric.Data;
using Microsoft.ServiceFabric.Data.Collections;
 [Route("api/[controller]")]
public class VoteDataController : Controller
{
private readonly IReliableStateManager stateManager;
public VoteDataController(IReliableStateManager stateManager)
{
this.stateManager = stateManager;
}
// GET api/VoteData [HttpGet]
public async Task<IActionResult> Get()
{
CancellationToken ct = new CancellationToken();
IReliableDictionary<string, int> votesDictionary =
await this.stateManager.GetOrAddAsync<IReliableDictionary<string,
int>>("counts");
using (ITransaction tx = this.stateManager.CreateTransaction())
{
var list = await votesDictionary.CreateEnumerableAsync(tx); var enumerator =
list.GetAsyncEnumerator();
List<KeyValuePair<string, int>> result = new List<KeyValuePair<string, int>>();
while (await enumerator.MoveNextAsync(ct))
{
result.Add(enumerator.Current);
}
return this.Json(result);
}
}
// PUT api/VoteData/name [HttpPut("{name}")]
public async Task<IActionResult> Put(string name)
```

```
{
IReliableDictionary<string, int> votesDictionary = await
this.stateManager.GetOrAddAsync<IReliableDictionary<string, int>>("counts");
using (ITransaction tx = this.stateManager.CreateTransaction())
{
await votesDictionary.AddOrUpdateAsync(tx, name, 1, (key, oldvalue) =>
oldvalue + 1); await tx.CommitAsync();
}
return new OkResult();
}
// DELETE api/VoteData/name [HttpDelete("{name}")]
public async Task<IActionResult> Delete(string name)
{
IReliableDictionary<string, int> votesDictionary = await
this.stateManager.GetOrAddAsync<IReliableDictionary<string, int>>("counts");
using (ITransaction tx = this.stateManager.CreateTransaction())
{
if (await votesDictionary.ContainsKeyAsync(tx, name))
{
await votesDictionary.TryRemoveAsync(tx, name); await tx.CommitAsync();
return new OkResult();
}
else
{
return new NotFoundResult();
}
}
}
}
}
```

Step 4: Publish Service Fabric application

a. In the solution Explorer Right click on the Voting and select Publish. The Publish dialog box appears.
b. To open the service fabric explorer, open the Browser and paste the following IP address http://localhost:19080/
c. Go to cluster >Application >Voting Type >fabric:/Voting >fabric:/VotingWeb > 8482ef73-476d- 45a6-aca9-b33d75575855(partition) >_Node_0 Instance
d. You will see the endpoint in the address section click on that address to run your app

PRACTICAL 2

Develop Spring Boot API

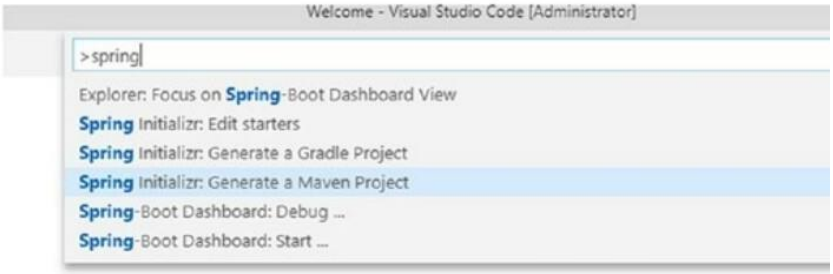
Setting up the Development Environment

Let's set up the development environment.

1. Install Visual Studio 2017.
2. Install the Microsoft Azure Service Fabric SDK.
3. Install Visual Studio Code.
    a. Install Spring Boot Extensions Pack.
    b. Install Java Extensions Pack.
    c. Install Maven for Java.
4. Make sure that the Service Fabric Local cluster is in a running state.
5. Install Docker Desktop.
6. Access the Azure container registry.

Develop a Spring Boot API

Now it's time to get started on the application

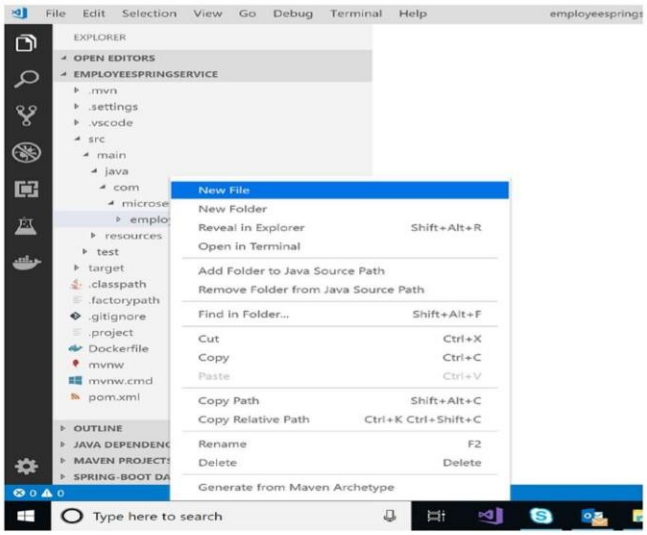


1. Launch Visual Studio Code as an administrator.
2. Press Ctrl+Shift+P to open the command palette.
3. Enter spring in the command palette, and choose Spring Initializr: Generate a maven project
4. Choose Java for Specify Project Language.
5. Enter com.microservices in the input group ID for your project.
6. Enter employeespringservice in the input artifact ID for your project.
7. Choose the latest Spring boot version.
8. Choose the following dependencies.
    a. DevTools
    b. Lombok
    c. Web
    d. Actuator
9. Choose the path where you want to save the solution.
10. Right-click the EMPLOYEESPRINGSERVICE folder under src ➤ main ➤ java ➤ com ➤microservices, as shown in Figure 3-15, and click Add File. Add a new file
11. Name the file Employee.Java and add the following code. (This is the definition of the employee object; we kept it simple by having only three

properties to represent an employee.) package com.microservices.employeespringservice;

```
import
lombok.AllArgsConstructor;
import
lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.Setter;
/*** Employee
***/ @Getter
@Setter
@EqualsAndHashCode
@AllArgsConstructor
public class Employee { private String firstName; private String lastName;
private String ipAddress; }
```

12. Now let's create an employeeservice that returns an employee's information. Right-click the employeespringservice folder and add a file named EmployeeService.java. Add the following code to it.

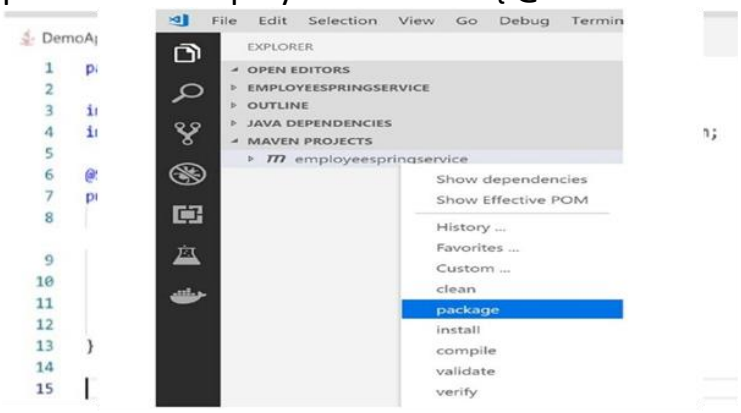



```
package
com.microservices.employeespringservice;
import java.net.InetAddress;
import java.net.UnknownHostException;
import
org.springframework.stereotype.Service;
/**
* EmployeeService
*/
@Service
public class EmployeeService {
```

```
public Employee GetEmployee(String
firstName, String lastName){
String
ipAddress; try {
getHostAddress().toString();
} catch
(UnknownHostException e) {
ipAddress = e.getMessage();
}
Employee employee = new
Employee(firstName,
lastName,ipAddress);
return employee;
}
}
```

13. Now let's create an employee controller that invokes the employee service to return the details of an employee. Right-click the employeespringservice folder and add a file named EmployeeController.

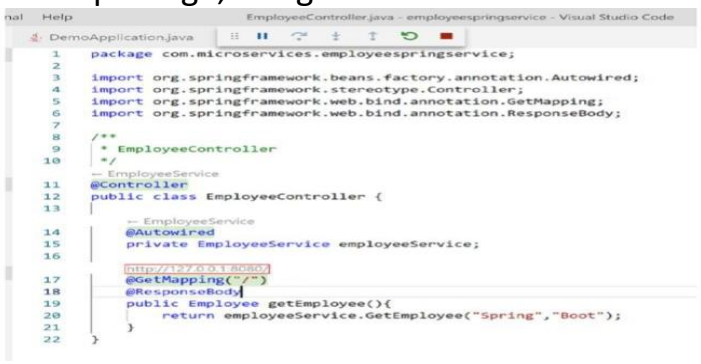
```
package com.microservices.employeespringservice;
import
org.springframework.beans.factory.annotation.Autowi
red; import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.GetMappin
g; import
org.springframework.web.bind.annotation.ResponseBody;
/**
* EmployeeController
*/ @Controller
public class EmployeeController { @Autowired
```

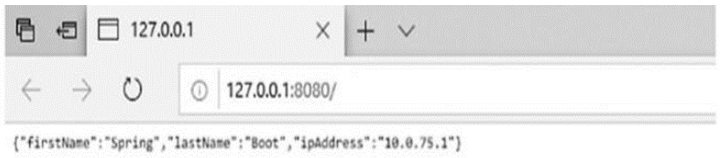


EmployeeService

employeeService;
@GetMapping("/")
@ResponseBody
public Employee getEmployee(){
return employeeService.GetEmployee("Spring","Boot");
}
}
Now you are ready for a simple REST-based service that returns employee information. Visual Studio Code has some cool features to run Spring Boot applications, and we are going to use the same.

1. Open DemoApplication.java and once you open the file, you see the option to run or debug the application,Please note that this may take time as Visual Studio automatically downloads the dependencies.
2. Click Run and open the Controller class. You see the URL where your controller service is hosted.
3. Click the URL and you see the output show in your default browser.
4. Right- click employeespringservice under Maven Projects. Click package,This generates the JAR file.





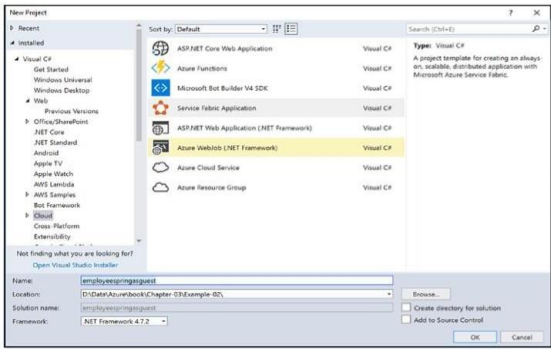
Now you have a simple Spring Boot-based REST API,and you have generated a JAR file. We will now deploy it to Service Fabric as a guest executable. To deploy, we will use Visual Studio 2017.

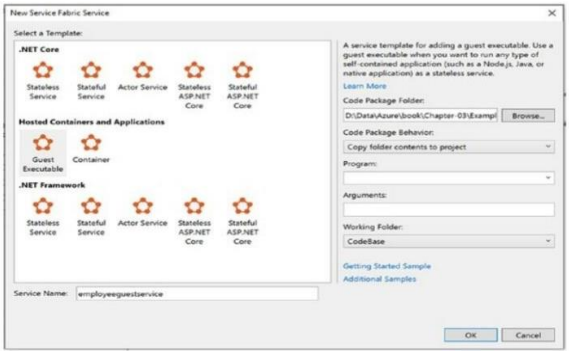
Deploy a Spring Boot Service as a Guest Executable

After executing all the steps in the previous section, your development is complete. Please follow the steps in this section to deploy the developed Spring Boot application as a guest executable. This shows that it is possible to

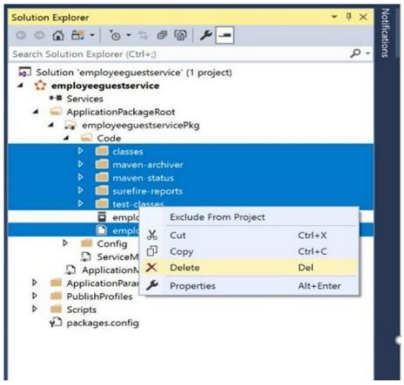
host a non-Microsoft stack application on a Service Fabric cluster by using a guest executable programming model. Service Fabric considers guest executables a stateless service.

1. Launch Visual Studio 2017 as an administrator.
2. Click File ➤ New Project ➤ Select Cloud ➤ Service Fabric Application.
3. Name the application employeespringasguest

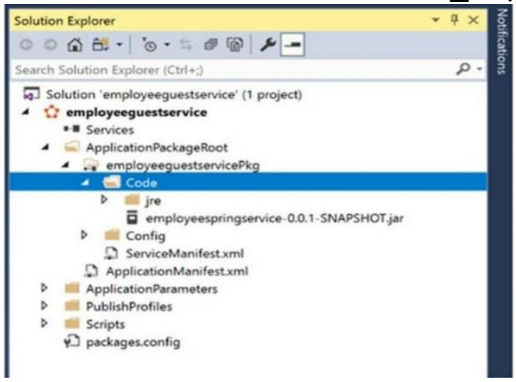




4. In New Service Fabric Service, select the following
   a. Service Name: employeeguestservice
   b. Code Package Folder: Point to the target folder in which Visual Studio Code generated the JAR file for the Spring Boot service.
   c. Code Package Behavior: Copy folder contents to folder
   d. Working Folder: CodeBase
5. Delete the selected files from the Code folder.

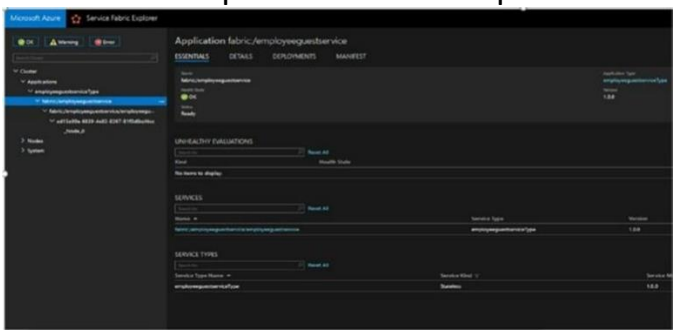


6. We also need to upload the runtime to run the JAR. Generally, it resides in the JDK installation folder (C:\ java-1.8.0-openjdk-1.8.0.191 1.b12.redhat.windows.x86_64). Paste it in the Code folder.

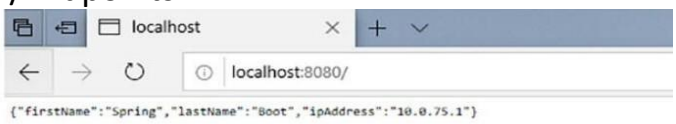


7. Open ServiceManisfest.xml and set the following values.

```
<EntryPoint>
<ExeHost>
<Program>jre\bin\java.exe</Program>
<Arguments>-jar
..\..\employeespringservice-0.0.1-
SNAPSHOT.jar</Arguments>
<WorkingFolder>CodeBase</WorkingFolder>
<!-- Uncomment to log console output (both stdout and stderr)
to one of the service's working directories. -->
<!-- <ConsoleRedirection FileRetentionCount="5" FileMaxSizeInKb="2048"/> --
>
</ExeHost>
</EntryPoint>
</CodePackage>
<Resources>
<Endpoints>
<!-- This endpoint is used by the communication listener to obtain the port on
which to listen. Please note that if your service is partitioned, this port is
shared with replicas of different partitions that are placed in your code.
```



```
<Endpoint
Name="employeeguestserviceTypeEndpoint"
Protocol="http" Port="8080" Type="Input" />
</Endpoints>
```



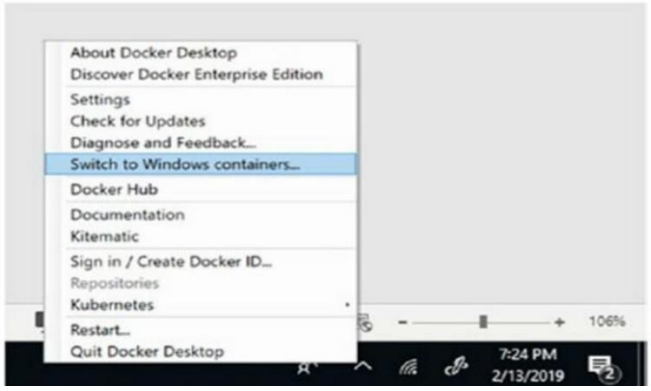

```
</Resources>
```

8. Make sure that the local Service Fabric cluster is up and running. Click F5. Browse the Service Fabric dashboard. The default URL is http://localhost:19080/Explorer/index.html. You see that your service is deployed.
9. Browse http://localhost:8080 to access your service. In servicemanifest.xml, we specified the service port as 8080; you can browse the same on 8080.

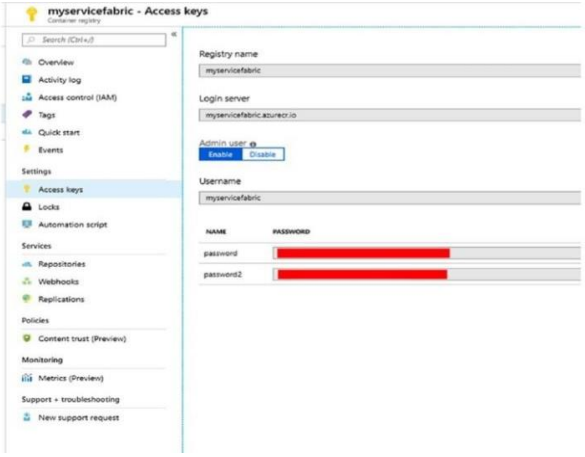
Deploy a Spring Boot Service as a Container

So far, we have deployed the service as a guest executable in Service Fabric. Now we will follow the steps to deploy the Spring service as a container in Service Fabric. This explains that in addition to creating stateful and stateless services, Service Fabric also orchestrates containers like any other orchestrator, even if the application wasn't developed on a Microsoft stack.

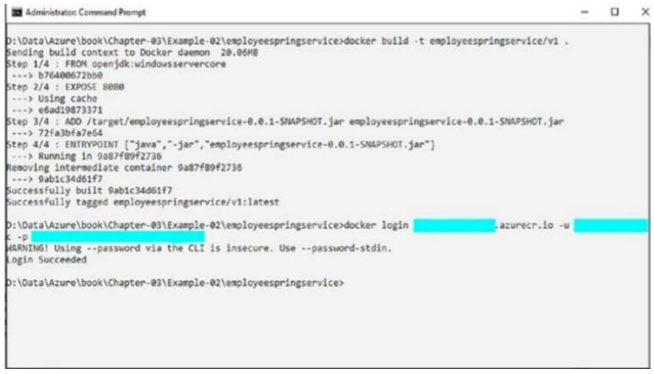
1. Open Visual Studio Code. Open the folder where employeespringservice exists. Open the Docker file.
2. Make sure that the name of the JAR file is correct.
3. Select Switch to Windows container… in Docker Desktop.



4. Create the Azure Container Registry resource in the Azure portal. Enable the admin user.



5. Open the command prompt in Administrative Mode and browse to the directory where the Docker file exists.
6. Fire the following command, including the period at the end. (This may take time because it downloads the Window Server core image from the Docker hub) docker build -t employeespringservice/v1 .



Now the container image is available locally. You have to push the image to Azure Container

Registry.

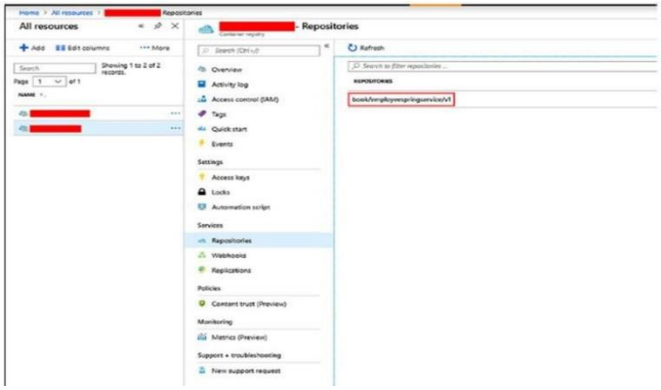
1. Log in to Azure Container Registry using the admin username and password. Use the following command . docker login youracr.azurecr.io -u yourusername -p yourpassword
2. Fire the following commands to upload the image to ACR. docker tag employeespringservice/v1 youracr.azurecr.io/book/

employeespringservice/v1
docker push
myservicefabric.azurecr.io/book/
employeespringservice/v1

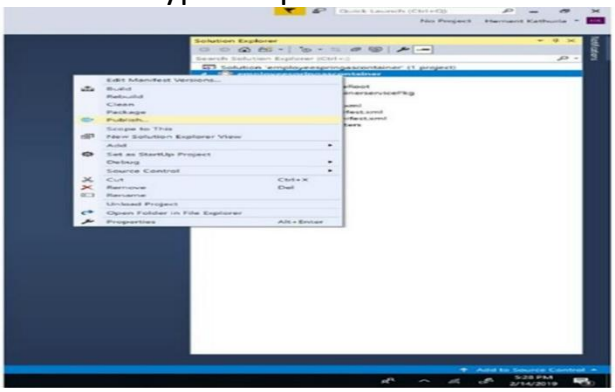
3. Log in to the Azure portal and check if you can see your image in Repositories.Since the container image is ready and uploaded in Azure Container Registry, let’s create a Service Fabric project to deploy the container to the local Service Fabric cluster.

1. Launch Visual Studio 2017 as an administrator.

2. Click File ➤ New Project ➤ Select Cloud ➤ Service Fabric Application.
3. Name the application employeespringascontainer.

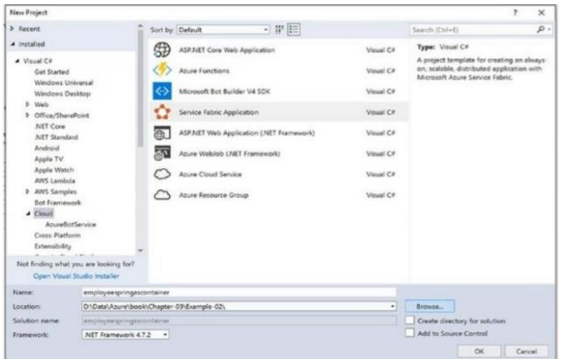


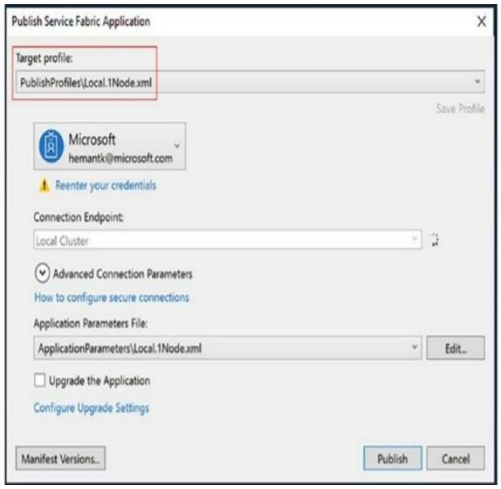
4. In New Service Fabric Service, select the following.
   a. Service Name: employeecontainerservice
   b. Image Name: youracr.azurecr.io/book/employeespringservice/v1
   c. User Name: Your username in the Azure Container Registry
   d. Host Port: 8090
   e. Container Port: 8080
5. Once the solution is created, open the ApplicationManifest.xml. Specify the right password for the admin user. (Since this is a sample, we kept the password unencrypted; for real-word applications you have to encrypt the password



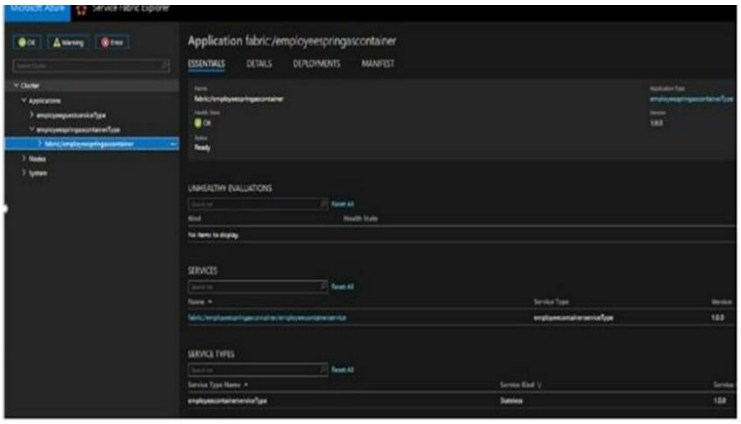
Now we are ready to build and deploy the container to the local Service Fabric cluster. Since we have given the user information to download the image from Azure Container Registry, Visual Studio downloads and deploy the container to the local services fabric cluster.

1. Right-click the Service Fabric project and publish.
2. Select the local cluster profile to publish to the local Service Fabric cluster. To deploy to Azure, select the cloud profile. Make sure that the Service Fabric cluster is up and ready in your subscription





3. Browse the Service Fabric dashboard. The default URL is http://localhost:19080/Explorer/index.html. Your service is deployed.



4. Browse to http://localhost:8090/ to access your service. You get the response which is served from the container run by Service Fabric.

PRACTICAL NO:3

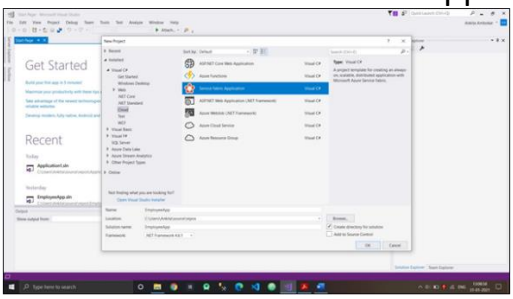
Create an Asp.Net Web Core Api and Configure Monitoring.

Setting up the Development Environment Let's set up.

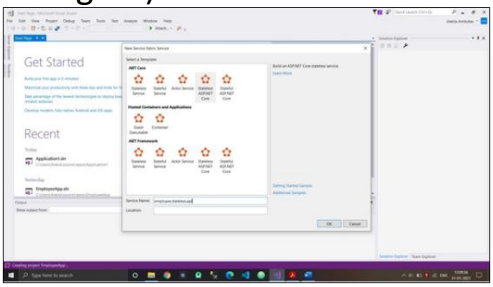
1. Install Visual Studio 2017.
2. Install the Microsoft Azure Service Fabric SDK.
3. Create the Translator Text API in your Azure subscription and make a note of the access key.
4. Create an empty Azure SQL Database and keep the connection string with SQL Authentication handy.
5. Make sure that the Service Fabric local cluster on Windows is in a running state.
6. Make sure that the Service Fabric Azure cluster on Windows is in a running state.

Create an ASP.NET Core Web API Now let's start the API.

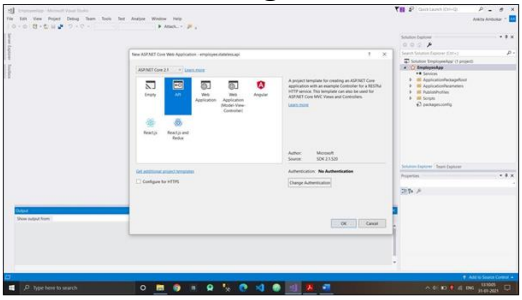
1. Launch Visual Studio 2017 as an administrator.
2. Create a project by selecting File →New → Project.
3. In the New Project dialog, choose Cloud →Service Fabric Application.
4. Name the Service Fabric application EmployeeApp and click OK.



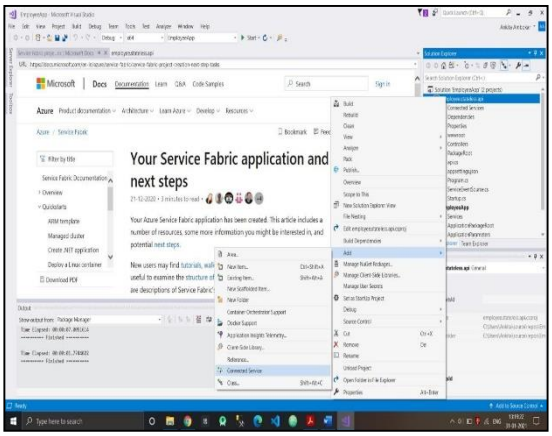
5. Name the stateless ASP.NET Core service Employee.Stateless.Api (as seen in Figure) and click OK.



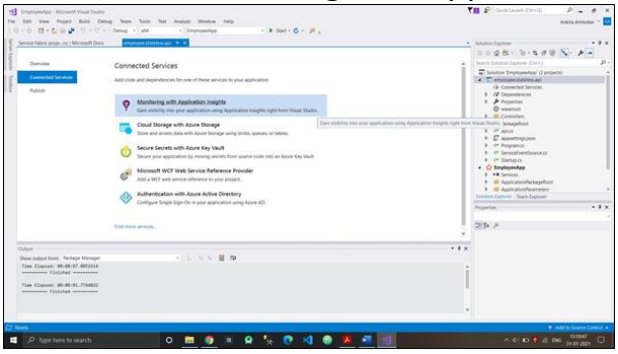
6. Choose the API and click OK. Make sure that ASP.NET Core 2.2 is selected, as shown in Figure.



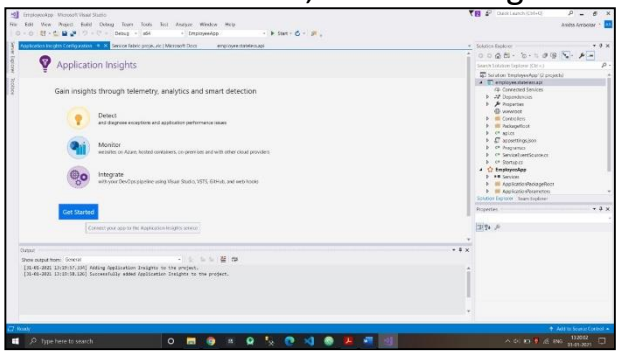
7. Right-click the employee.stateless.api project and select Add →Connected Service.



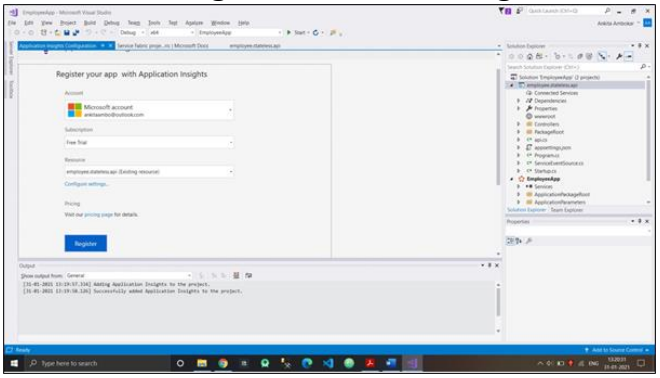
8. Choose Monitoring with Application Insights, as seen in Figure.



9. Click Get Started, as seen in Figure.

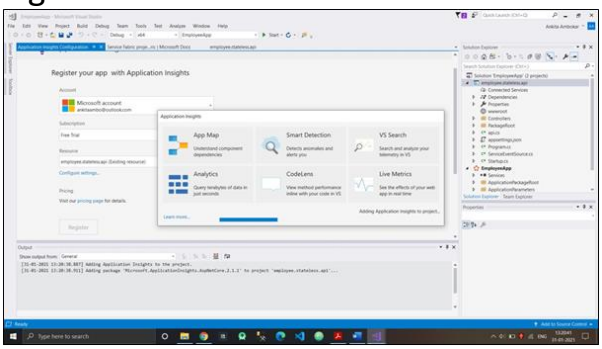


Choose the right Azure subscription and Application Insights resource. Once done, click Register, as seen in Figure.

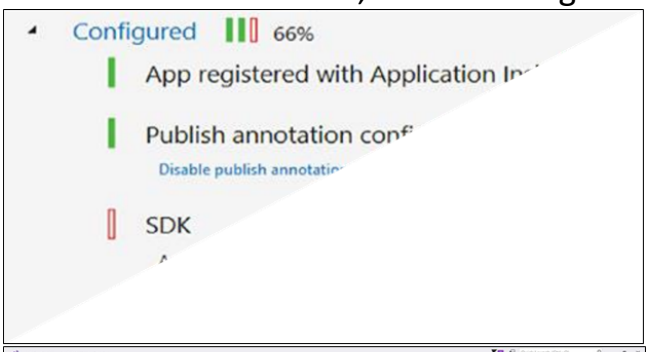


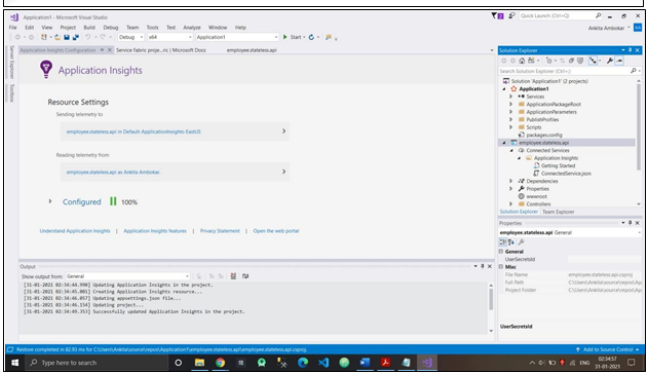
It takes a few minutes to create the Application Insights resource in your Azure subscription. During the registration process, you see the screen shown in

Figure.



10. Once the Application Insights configuration is complete, you see the status as 100%. If you see the Add SDK button (as shown in Figure), click it to achieve 100% status, as seen in Figure.



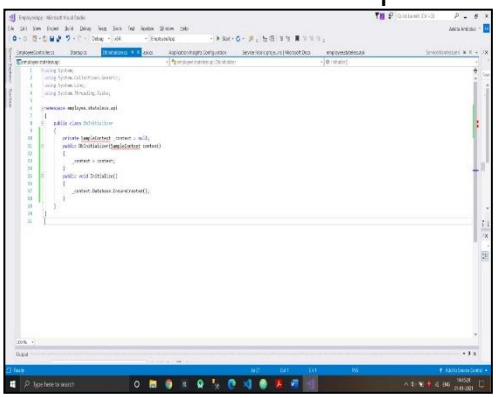


11. To confirm the Application Insights configuration, check the instrumentation key in appsettings.json.
12. Right-click the employee.stateless.api project to add dependencies for the following NuGet packages.
    a. Microsoft.EntityFrameworkCore.SqlServer
    b. Microsoft.ApplicationInsights.ServiceFabric.Native
    c. Microsoft.ApplicationInsights.AspNetCore

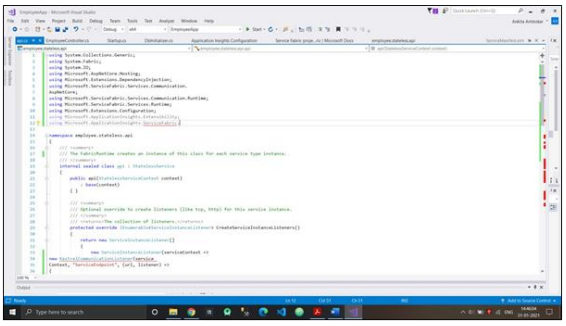
We are done with the configuration. Now let's add EmployeeController.

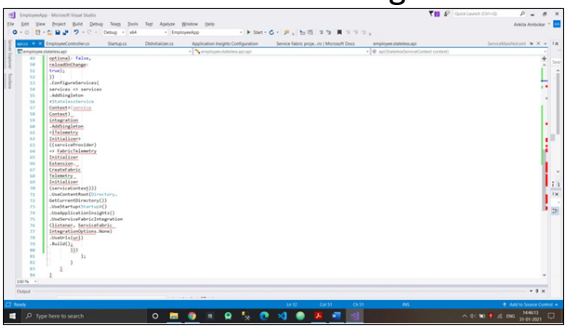
1. Right-click the employee.stateless.api project and a folder called Models. Add the following classes from the sources folder.
    a. AppSettings.cs
    b. Employee.cs
    c. SampleContext.cs
    d. TranslationResponse.cs

2. Right-click the employee.stateless.api project and add a file named DbInitializer.cs. Replace that content with the following content

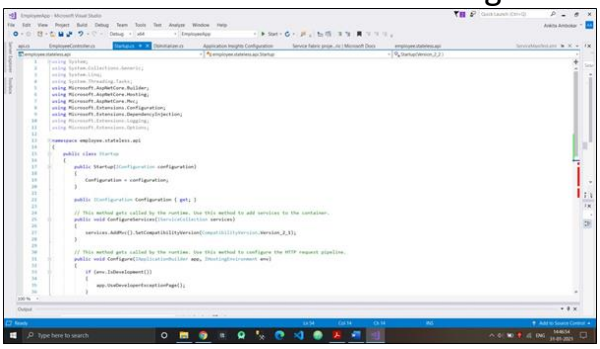


3. Open Api.cs and replace the contents of the CreateServiceInstanceListeners method with the following content.

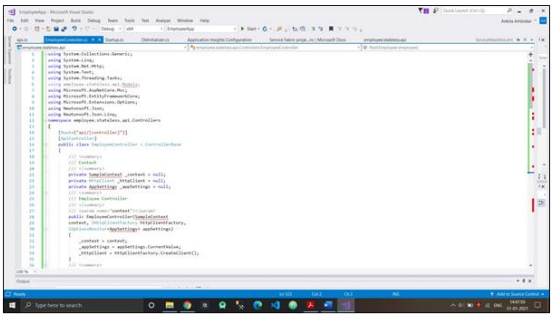


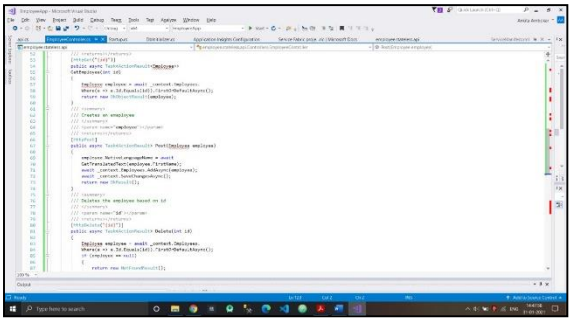


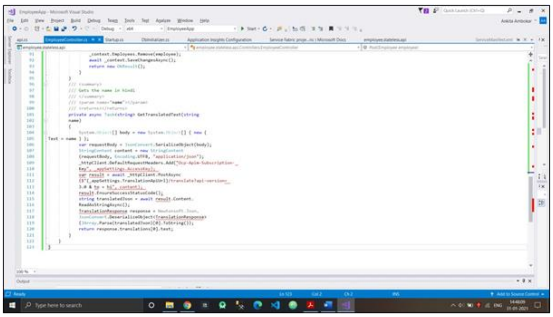
4. Open Startup.cs and replace the contents of the ConfigureServices method with the following content.



5. Right-click the controller folder in the employee.stateless.api project and add a controller called EmployeeController.cs. Replace that content with the following content.







6. Open AppSettings.json and make sure that the content looks similar to your connection strings.

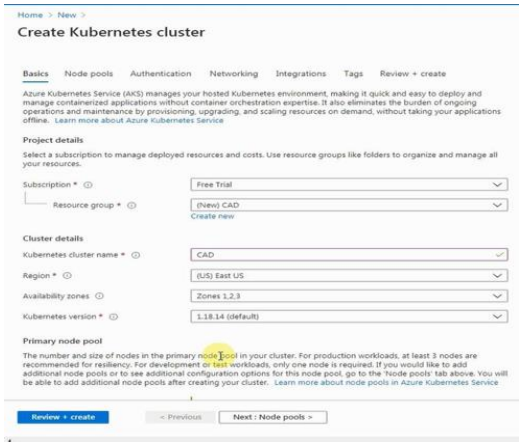
```
{
"ConnectionStrings": {
"DefaultConnection": "Server=<<YOUR_SERVER>>;Database=
<<YOUR_DATABASE>>;User ID=<<USER_ID>>;Password=<<PASSWORD>>
;Trusted_Connection=False;Encrypt=True;MultipleActiveResult
Sets=True;"
},
"AppSettings": {
"TranslationApiUrl": "https://api.cognitive.
microsofttranslator.com",
"AccessKey": "<<YOUR ACCESS KEY TO TRANSLATION API>>"
},
"Logging": {
"LogLevel": {
"Default": "Warning"
}
},
"AllowedHosts": "*",
"ApplicationInsights": {
"InstrumentationKey": "<<YOUR INSTRUMENTATION KEY OF YOUR APP
INSIGHTS RESOURCE>>"
}
```

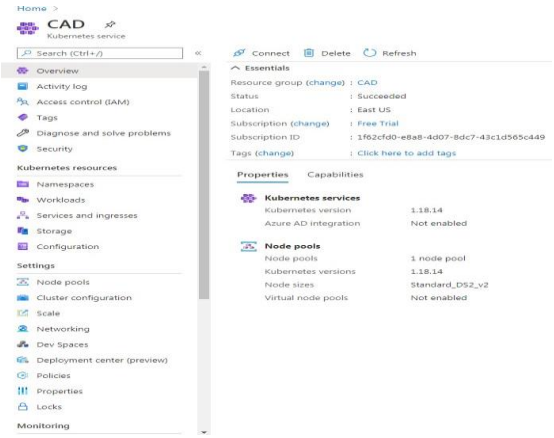
Practical No.4

A. Create an Azure Kubernetes Service Cluster

Steps –

1. Sign in Azure Portal (https://portal.azure.com/)
2. Create Resource by clicking on Create Resource option.
3. Select Kubernetes Services
4. Enter the subscription, resource group, kubernetes, Cluster name, Region, Kubernetes version,& DNS Name prefix
5. Click on Review + Create Button
6. Click on Create Button





B. Enable Azure Dev Space on an AKS Cluster

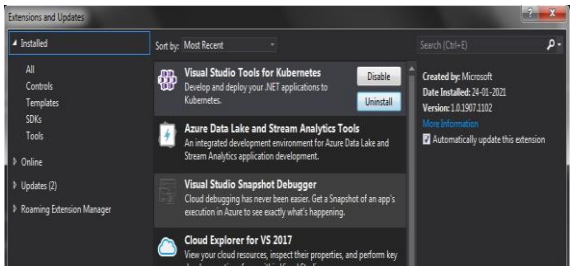
Steps –

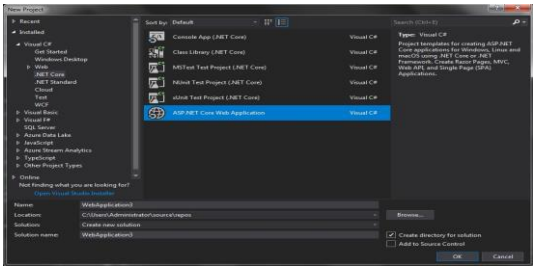
1. Open Kubernetes Services
2. Click on Dev Space
3. Install Azure CLI
4. Open CMD
5. Enter command AZ LOGIN
6. Enter command az aks use-dev-spaces –n CAD–g CAD
7. Install Azure Dev Space

C. Configure Visual Studio to work with an azure kubernetes services Cluster

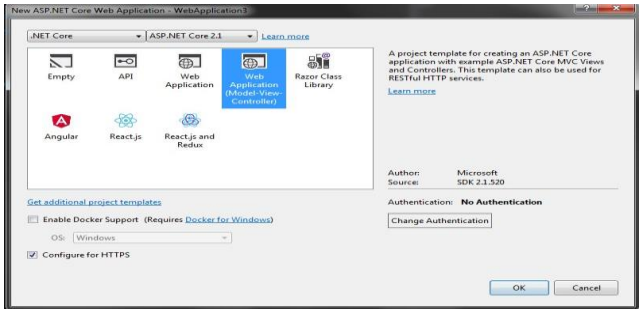
Steps –

1. Install Visual Studio Enterprise 2017
2. Open Visual Studio 2017 >Tool > Extensions & Update
3. Search for Kubernetes Services tool
4. Click on Download
5. Close visual studio
6. click on Modify(install kubernetes tool extension)
7. Open Visual studio
8. Click on new project
9. Select Asp.net Core Web Application > click on Ok

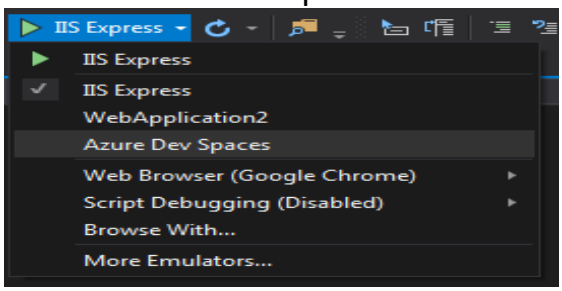




10. Select Model View Control > Click on OK



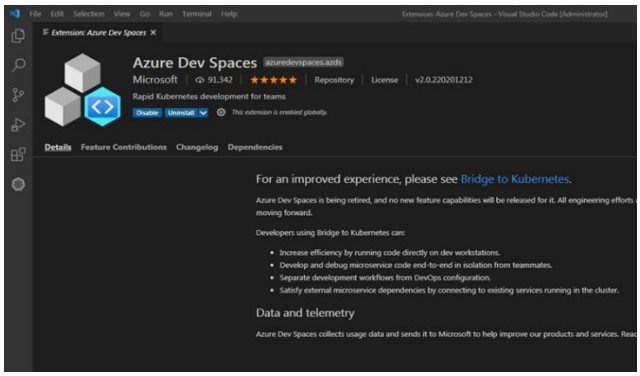
11. Click On Down Arrow of IIS Express
12. We can see option of Azure Dev Space



D. Configure Visual Studio Code to work with an azure kubernetes services Cluster

Steps –

1. Install visual studio code
2. Open extension window
3. Search for Azure Dev Space
4. Click Install



5. Install Azure CLI (https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli- latest)
6. Open cmd
7. Enter command az Login

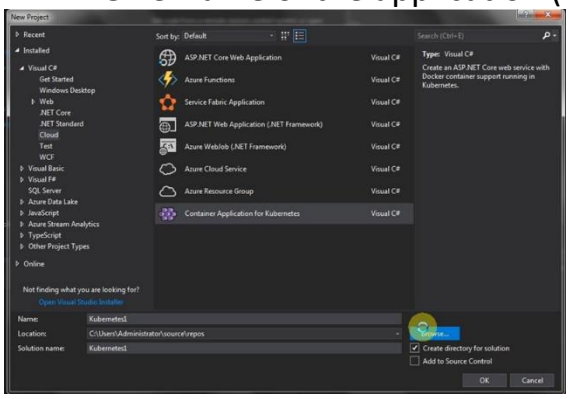
8. Enter Command to install AZDS utility az aks use-dev-spaces –n CAD –g CAD

E. Deploy Application on AKS

1. Core web API

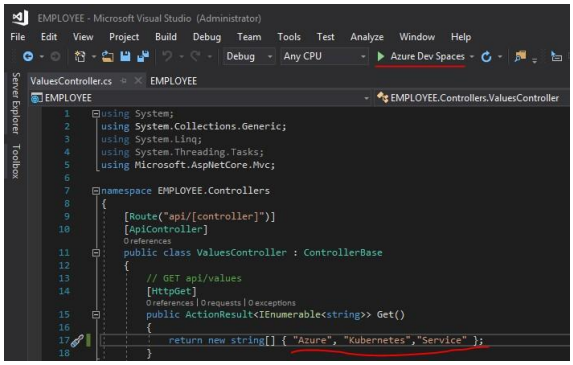
Steps –

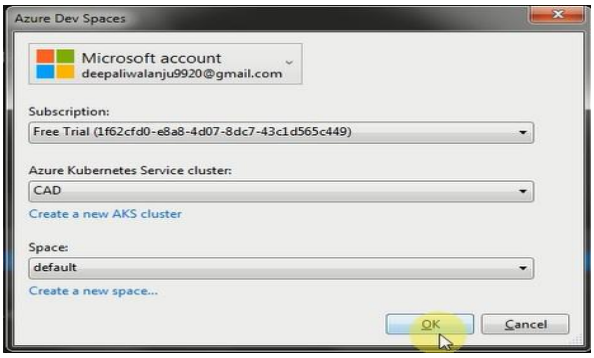
1. Open Visual Studio
2. Click on New project
3. Select Cloud Container Application for Kubernetes
4. Give Name of the application (Employee) & click on Ok

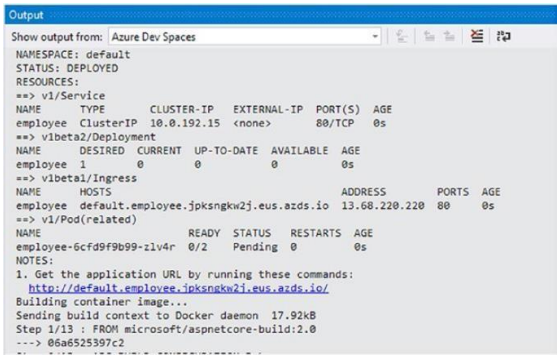


5. Select API option & click on OK button

6. Make sure Azure Dev Space is selected in menu.
7. Open ValuesController.cs file
8. Edit line number 16 as shown in the screen shot
9. Save changes & press F5 button to run the application
10. Select cluster name & space > click on OK button







2. Node.js API

Steps –

1. Create MYAPP folder
2. Open visual studio code
3. Open folder MYAPP in visual studio code
4. Create new file server.js & pacakge.json
5. From windows explorer create PUBLIC folder in MYAPP folder
6. Now create new file using public folder by using visual studio code
7. Create file index.html & app.css
8. Open command palette from view menu (ctrl+shif+p)
9. Enter azure dev space & click on it
10. Click on configure
11. Click DEBUG icon on left & then click on LAUNCH SERVER(AZDS)
12. OUTPUT will display in Output window

Server.js

```
var express = require('express'); var app = express();
app.use(express.static( dirname + '/public')); app.get('/', function (req, res) {
res.sendFile( dirname + '/public/index.html');
});
app.get('/api', function (req, res) { res.send('Hello from webfrontend');
});
var port = process.env.PORT || 80;
var server = app.listen(port, function () { console.log('Listening on port ' + port);
});
```

```
process.on("SIGINT", () => { process.exit(130 /* 128 + SIGINT */);
});
process.on("SIGTERM", () => { console.log("Terminating..."); server.close();
});
```

Package.json

```
{
"name": "webfrontend", "version": "0.1.0", "devDependencies": { "nodemon":
"^1.18.10"
},
"dependencies": { "express": "^4.16.2", "request": "2.83.0"
},
"main": "server.js", "scripts": {
"start": "node server.js"
}
}
```

Index.html

```
<!doctype html>
<html ng-app="myApp">
<head>
<script src="https://ajax.googleapis.com/ajax/libs/
angularjs/1.5.3/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/ libs/angular.js/1.5.3/angular-
route.js"></script>
<script src="app.js"></script>
<link rel="stylesheet" href="app.css">
<link href="https://maxcdn.bootstrapcdn.
com/bootstrap/3.3.6/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-1q8mTJOASx8j1Au
+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
<!-- Uncomment the next line -->
<!-- <meta name="viewport" content="width=devicewidth, initial-scale=1"> -->
</head>
<body style="margin-left:10px; margin-right:10px;">
<div ng-controller="MainController">
<h2>Server Says</h2>
<div class="row">
<div class="col-xs-8 col-md-10">
<div ng-repeat="message in messages track by $index">
<span class="message">{{message}}</ span>
```

```
</div>
</div>
<div class="col-xs-4 col-md-2">
<button class="btn btn-primary" ng-click=" sayHelloToServer()">Say It
Again</button>
</div>
</div>
</div>
</body>
</html>
```

APP.css

```
.message {
font-family: Courier New, Courier, monospace; font-weight: bold;
}
```

OUTPUT

The Debug console shows the log output.

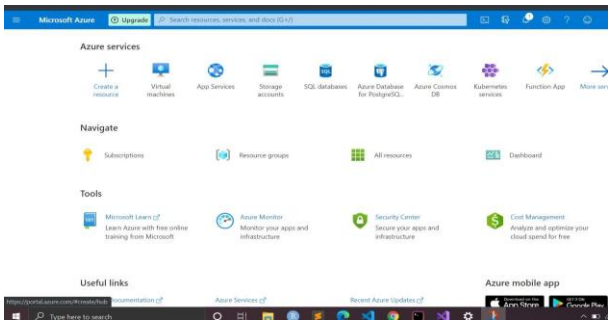
> Executing task: C:\Program Files\Microsoft SDKs\Azure\Azure Dev Spaces CLI (Preview)\azds.exe up --port=50521:9229 --await-exec --keep-alive < Synchronizing files...4s
Using dev space 'new01' with target 'kuber01' Installing Helm chart...2s
Waiting for container image build...29s Building container image...
Step 1/8 : FROM node:lts Step 2/8 : ENV PORT 80 Step 3/8 : EXPOSE 80
Step 4/8 : WORKDIR /app Step 5/8 : COPY package.json . Step 6/8 : RUN npm install Step 7/8 : COPY . .
Step 8/8 : CMD ["npm", "start"] Built container image in 45s Waiting for container...52s
Service 'myapp' port 80 (http) is available via port forwarding at http://localhost:50764
Terminal will be reused by tasks, press any key to close it.

PRACTICAL 5
Create an AKS cluster

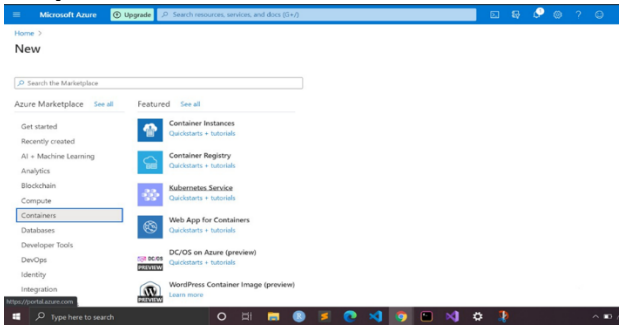
a. From the portal

Step 1: Sign in to the Azure portal at https://portal.azure.com.

Step 2: On the Azure portal menu or from the Home page, select Create a resource.



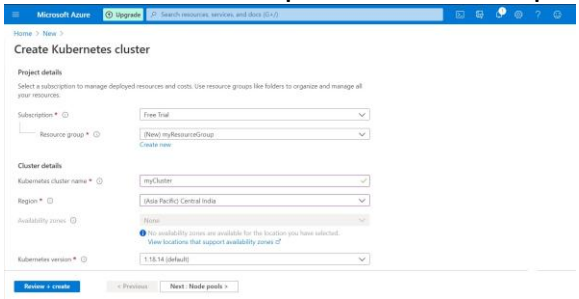
Step 3: Select Containers > Kubernetes Service.



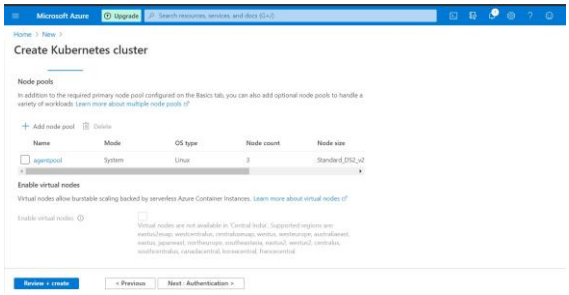
Step 4: On the Basics page, configure the following options:

- **Project details:** Select an Azure Subscription, then select or create an Azure Resource group, such as myResourceGroup.
- **Cluster details:** Enter a Kubernetes cluster name, such as myCluster. Select a Region and Kubernetes version for the AKS cluster.
- **Primary node pool:** Select a VM Node size for the AKS nodes. The VM size can't be changed once an AKS cluster has been deployed. - Select the number of nodes to deploy into the cluster. Set Node count to 1. Node count can be adjusted after the cluster has been deployed.

Select Next: Node pools when complete.

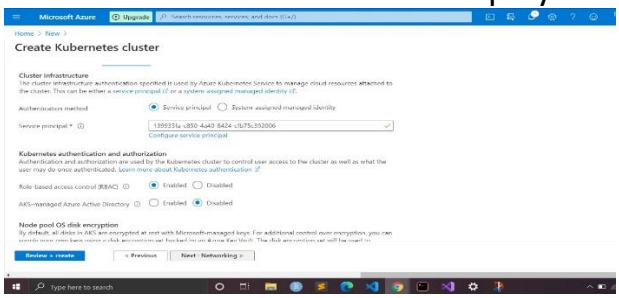


Step 5: On the Node pools page, keep the default options. At the bottom of the screen, click Next: Authentication.

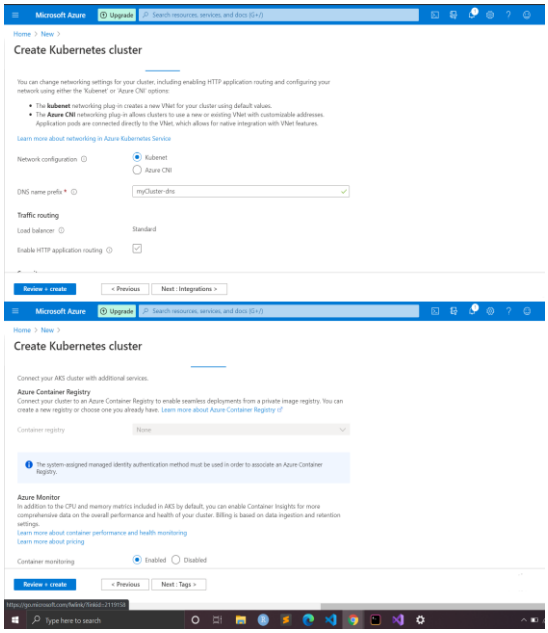


Step 6: On the Authentication page, configure the following options:

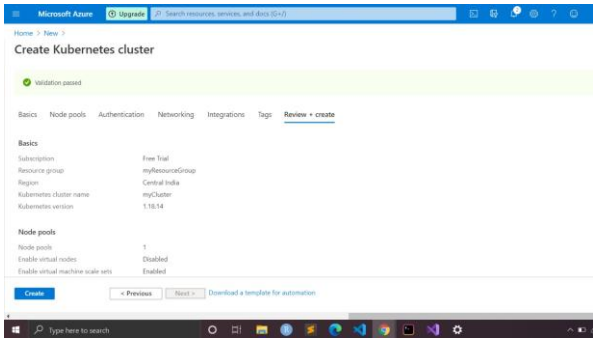
- Create a new service principal by leaving the Service Principal field with (new) default service principal. Or you can choose Configure service principal to use an existing one. If you use an existing one, you will need to provide the SPN client ID and secret.
- Enable the option for Kubernetes role-based access control (Kubernetes RBAC). This will provide more fine-grained control over access to the Kuber-netes resources deployed in your AKS cluster



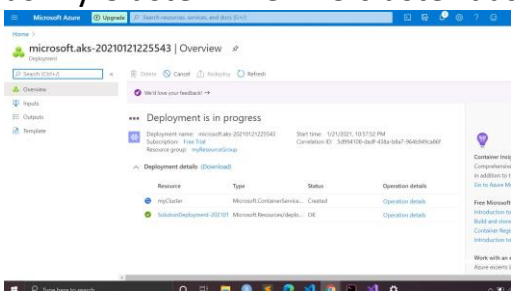
Step 7: By default, Basic networking, Integrations and tags is used, and Azure Monitor for containers is enabled.

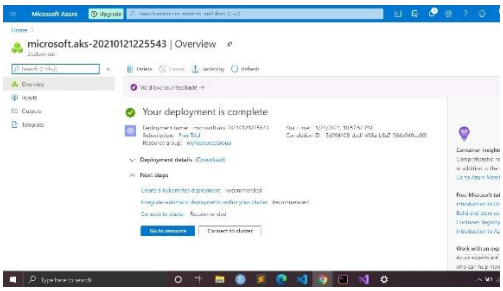


Click Review + create and then Create when validation completes.



Step 8: It takes a few minutes to create the AKS cluster. When your deployment is complete, click Go to resource, or browse to the AKS cluster resource group, such as myResourceGroup, and select the AKS resource, such as my Cluster. The AKS cluster dashboard is shown below:



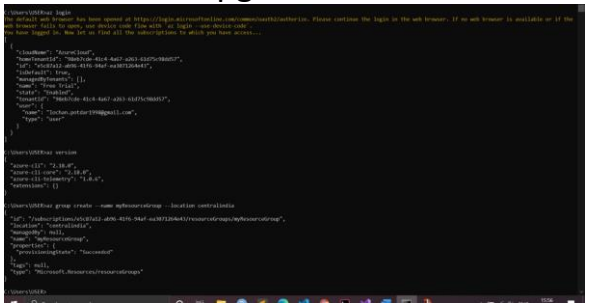


b. From azure CLI

Step 1: Install the Azure CLI to run CLI reference commands.

Step 2: Sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal.

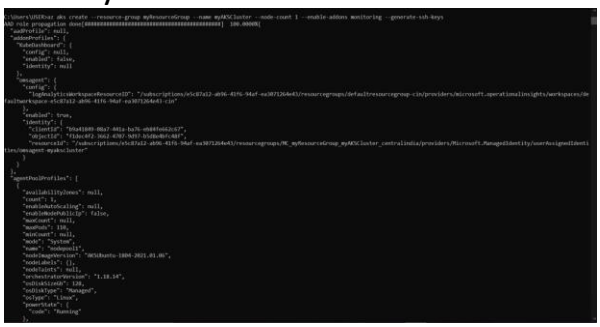
Step 3: Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.



Step 4: An Azure resource group is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are asked to specify a location. This location is where resource group metadata is stored, it is also where your resources run in Azure if you don't specify another region during resource creation. Create a resource group using the az group create command.

Step 5: The following example creates a resource group named myResourceGroup in the centralindia location. az group create --name myResourceGroup --location central india Output similar to the following example indicates the resource group has been created successfully:

Step 6: Use the az aks create command to create an AKS cluster. The following example creates a cluster named myAKSCluster with one node. This will take several minutes to complete. az aks create --resource-group myResourceGroup --name myAKSCluster --node-count 1 --enable-addons monitoring -- generate-ssh-keys



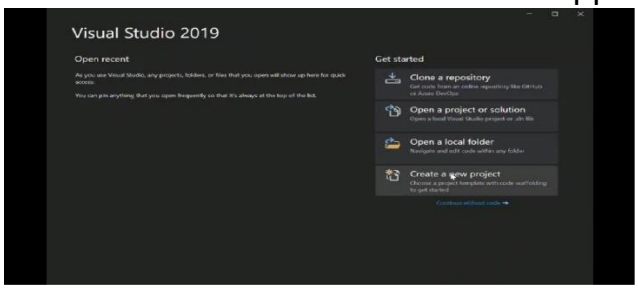
After a few minutes, the command completes and returns JSON-formatted information about the cluster.

Practical no 6

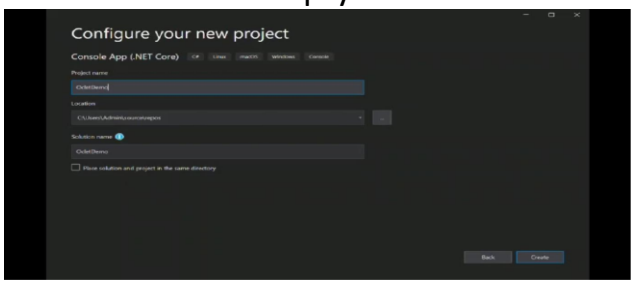
Create an Application Gateway Using Ocelot and Securing APIs with Azure AD

Steps:

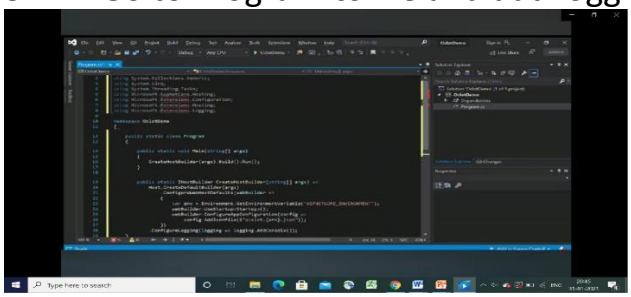
1. Create an ASP.NET Core Web Application Project.



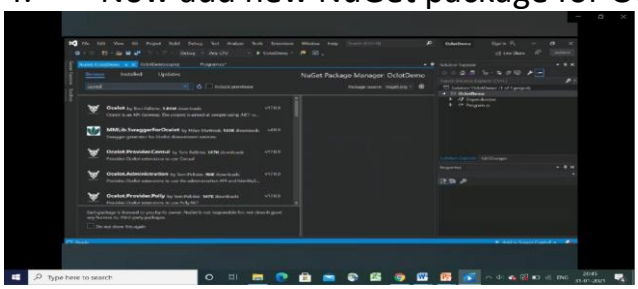
2. Create an empty ASP.NET Core 3.1 and and give a name of the Project.



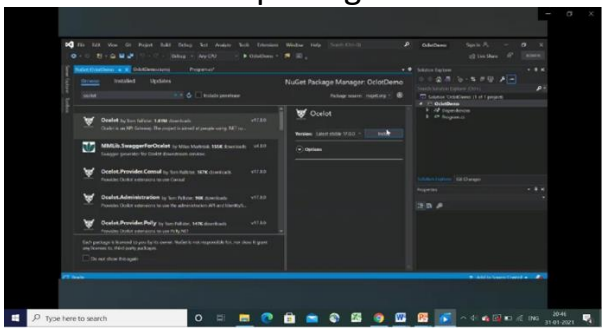
3. Go to Program.cs file and add logging code.



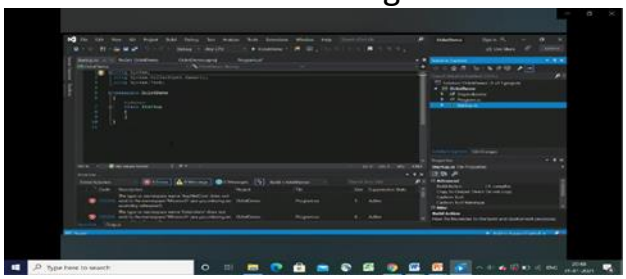
4. Now add new NuGet package for Ocelot



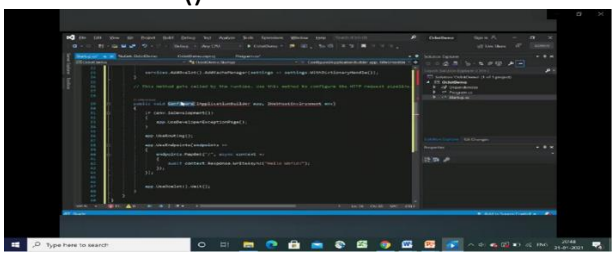
5. Click on Project -> Manage NuGet Package -> Click on Browse -> Search for ocelot package -> install.



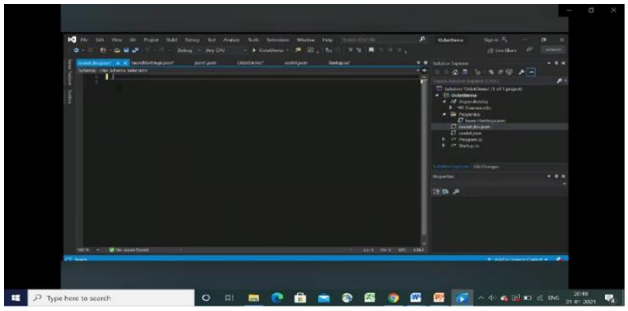
6. Once Ocelot Package is installed Go to Startup.cs



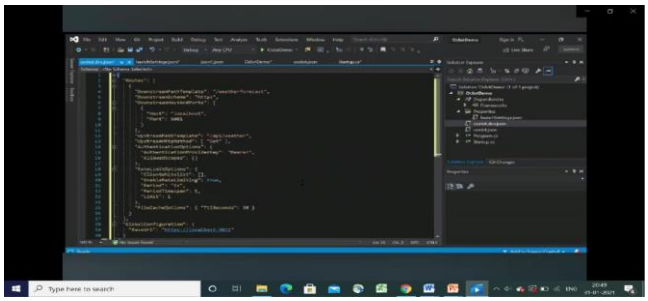
7. Now to Configure Ocelot add services.Addocelot() and app.UseOcelot() .Wait() code.



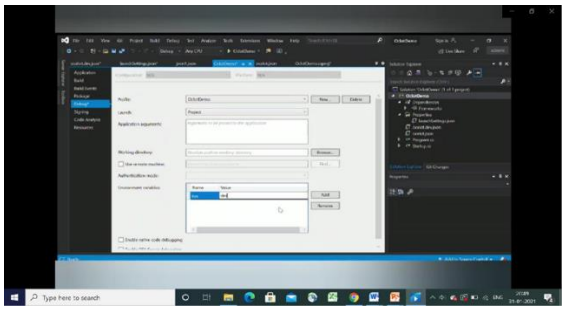
8. Now add the Ocelot JSON file

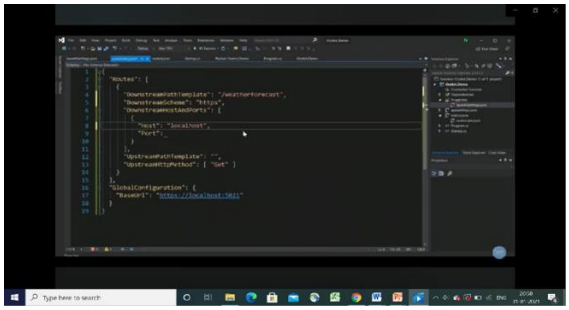


9. Click on Project name -> Add -> New Item -> JSON File ->(Give name)
10. Add JSON Code.

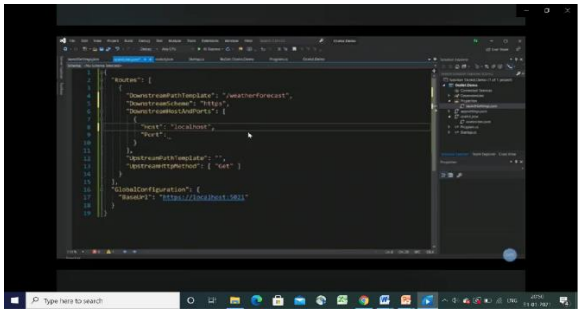


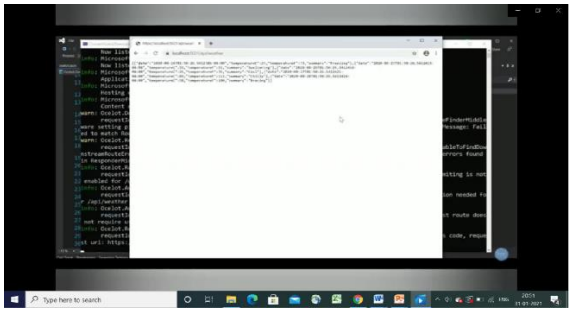
11. Create an Environmental Variable





12. Change localhost to 5020 and 5021.
13. Add Configuration to the system.





14. Run the project.

Code:

Program.cs file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
namespace Oclot.Demo
{
public static class Program
{
public static void Main(string[] args)
{
CreateHostBuilder(args).Build().Run(); }
public static IHostBuilder CreateHostBuilder(string[] args) =>
Host.CreateDefaultBuilder(args)
.ConfigureWebHostDefaults(webBuilder =>
{
var env =
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT");
webBuilder.UseStartup<Startup>();
webBuilder.ConfigureAppConfiguration(config =>
config.AddJsonFile($"ocelot.{env}.json"));
})
.ConfigureLogging(logging => logging.AddConsole());
}
}
```

Startup.cs file

```
namespace Oclot.Demo
{
```

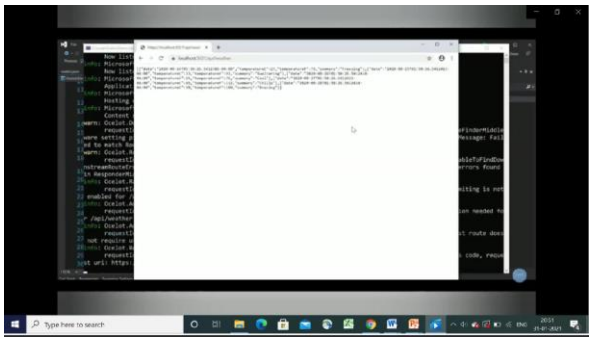
```
public class Startup
{
public void ConfigureServices(IServiceCollection services)
{
services.AddOcelot().AddCacheManager(settings =>
settings.WithDictionaryHandle());
}
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
if (env.IsDevelopment())
{
app.UseDeveloperExceptionPage();
}
app.UseRouting(); app.UseEndpoints(endpoints =>
{
endpoints.MapGet("/", async context =>
{
await context.Response.WriteAsync("Hello World!");
});
});
app.UseOcelot().Wait();
}
}
}
```

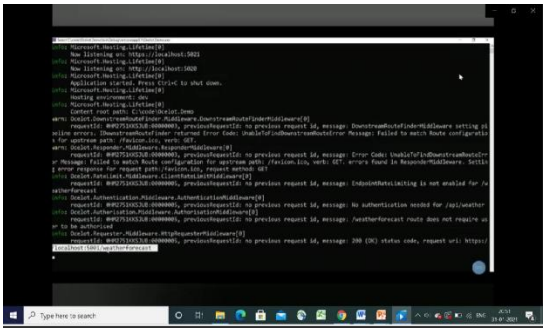
JSON file

```
{
"Routes": [
{
"DownstreamPathTemplate": "/weatherforecast", "DownstreamScheme":
"https", "DownstreamHostAndPorts": [
{
"Host": "localhost", "Port": 5001
} ],
"UpstreamPathTemplate": "/api/weather", "UpstreamHttpMethod": [ "Get" ],
"AuthenticationOptions": { "AuthenticationProviderKey": "Bearer",
"AllowedScopes": [] }, "RateLimitOptions": {
"ClientWhitelist": [], "EnableRateLimiting": true,
"Period": "5s", "PeriodTimespan": 1,
"Limit": 1 },
"FileCacheOptions": { "TtlSeconds": 30 }
```

```
}],
"GlobalConfiguration": { "BaseUrl": "https://localhost:5021"
}
}
```

OUTPUT





Practical no 7

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

Choosing a Data Store

There are many risks associated with embracing a 1.0-level technology. The ecosystem is generally immature, so support for your favorite things may be lacking or missing entirely. Tooling and integration and overall developer experience are often high-friction. Despite the long and storied history of .NET, .NET Core (and especially the associated tooling) should still be treated like a brand new 1.0 product.

One of things we might run into when trying to pick a data store that is compatible with EF Core is a lack of available providers. While this list will likely have grown by the time you read this, at the time this chapter was written, the following providers were available for EF Core:

- ☆ SQL Server
- ☆ SQLite
- ☆ Postgres
- ☆ IBM databases
- ☆ MySQL
- ☆ SQL Server Lite
- ☆ In-memory provider for testing
- ☆ Oracle

For databases that aren't inherently compatible with the Entity Framework relational model, like MongoDB, Neo4j, Cassandra, etc., you should be able to find client libraries available that will work with .NET Core. Since most of these databases expose simple RESTful APIs, you should still be able to use them even if you have to write your own client.

Because of my desire to keep everything as cross-platform as possible throughout this book, I decided to use Postgres instead of SQL Server to accommodate readers working on Linux or Mac workstations. Postgres is also easily installed on Windows.

Building a Postgres Repository

In order to get something running and focus solely on the discipline and code required to stand up a simple service, we used an in-memory repository that didn't amount to much more than a fake that aided us in writing tests.

In this section we're going upgrade our location service to work with Postgres. To do this we're going to create a new repository implementation that encapsulates the PostgreSQL client communication. Before we get to the implementation code, let's revisit the interface for our location repository.

Example 1. ILocationRecordRepository.cs

```
using System;
using System.Collections.Generic;
namespace StatlerWaldorfCorp.LocationService.Models { public interface
ILocationRecordRepository {
LocationRecord Add(LocationRecord locationRecord); LocationRecord
Update(LocationRecord locationRecord); LocationRecord Get(Guid memberId,
Guid recordId); LocationRecord Delete(Guid memberId, Guid recordId);
LocationRecord GetLatestForMember(Guid memberId);
ICollection<LocationRecord> AllForMember(Guid memberId);
}
}
```

The location repository exposes standard CRUD functions like Add, Update, Get, and Delete. In addition, this repository exposes methods to obtain the latest location entry for a member as well as the entire location history for a member.The purpose of the location service is solely to track location data, so you'll notice that there is no reference to team membership at all in this interface.

Creating a Database Context

The next thing we're going to do is create a database context. This class will serve as a wrapper around the base DbContext class we get from Entity Framework Core. Since we're dealing with locations, we'll call our context class LocationDbContext.

If you're not familiar with Entity Framework or EF Core, the database context acts as the gateway between your database-agnostic model class (POCOs, or Plain-Old C# Objects) and the real database. For more information on EF Core, check out Microsoft's documentation. The pattern for using a database context is to create a class that inherits from it that is specific to your model. In our case, since we're dealing with locations, we'll create a LocationDbContext class.

Example 2. LocationDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using StatlerWaldorfCorp.LocationService.Models; using
Npgsql.EntityFrameworkCore.PostgreSQL;
namespace StatlerWaldorfCorp.LocationService.Persistence
{
public class LocationDbContext : DbContext
{
public LocationDbContext( DbContextOptions<LocationDbContext> options) :
base(options)
```

```
{
}
protected override void OnModelCreating( ModelBuilder modelBuilder)
{
base.OnModelCreating(modelBuilder);
modelBuilder.HasPostgresExtension("uuid-ossp");
}
public DbSet<LocationRecord> LocationRecords {get; set;}
}
}
```

Here we can use the ModelBuilder and DbContextOptions classes to perform any additional setup we need on the context. In our case, we're ensuring that our model has the uuid-osspPostgres extension to support the member ID field.

Implementing the Location Record Repository Interface

Now that we have a context through which other classes can use to communicate with the database, we can create a real implementation of the ILocationRecordRepositoryinterface. This real implementation will take an instance of LocationDbContext as a constructor parameter. This sets us up nicely to configure this context with environment-supplied connection strings when deploying for real and with mocks or in-memory providers when testing. Example 3 contains the code for the LocationRecordRepositoryclass

Example 3. LocationRecordRepository.cs

```
using System; using System.Linq;
using System.Collections.Generic;
using StatlerWaldorfCorp.LocationService.Models;
namespace StatlerWaldorfCorp.LocationService.Persistence
{
public class LocationRecordRepository : ILocationRecordRepository
{
private LocationDbContext context;
public LocationRecordRepository(LocationDbContext context)
{
this.context = context;
}
public LocationRecord Add(LocationRecord locationRecord)
{
this.context.Add(locationRecord); this.context.SaveChanges();
return locationRecord;
}
```

```
public LocationRecord Update(LocationRecord locationRecord)
{
this.context.Entry(locationRecord).State = EntityState.Modified;
this.context.SaveChanges(); return locationRecord;
}
public LocationRecord Get(Guid memberId, Guid recordId)
{
return this.context.LocationRecords
.Single(lr => lr.MemberID == memberId &&
lr.ID == recordId);
}
public LocationRecord Delete(Guid memberId, Guid recordId)
{
LocationRecord locationRecord = this.Get(memberId, recordId);
this.context.Remove(locationRecord); this.context.SaveChanges();
return locationRecord;
}
public LocationRecord GetLatestForMember(Guid memberId)
{
LocationRecord locationRecord = this.context.LocationRecords.
Where(lr => lr.MemberID == memberId). OrderBy(lr => lr.Timestamp).
Last();
return locationRecord;
}
public ICollection<LocationRecord> AllForMember(Guid memberId)
{
return this.context.LocationRecords.
Where(lr => lr.MemberID == memberId). OrderBy(lr => lr.Timestamp).
ToList();
}
}
}
```

When we do an update, we need to flag the entity we're updating as a modified entry so that Entity Framework Core knows how to generate an appropriate SQL UPDATEstatement for that record.

If we don't modify this entry state, EF Core won't know anything has changed and so a call to SaveChangeswill do nothing.

The next big trick in this repository is injecting the Postgres-specific database context. To make this happen, we need to add this repository to the

dependency injection system in the ConfigureServices method of our Startup class.

Example 4. ConfigureServices method in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
services.AddEntityFrameworkNpgsql()
.AddDbContext<LocationDbContext>(options =>
options.UseNpgsql(Configuration));
services.AddScoped<ILocationRecordRepository, LocationRecordRepository>();
services.AddMvc();
}
```

First we want to use the AddEntityFrameworkNpgsql extension method exposed by the Postgres EF Core provider. Next, we add our location repository as a scoped service. When we use the AddScopedmethod, we're indicating that every new request made to our service gets a newly create instance of this repository.

Configuring a Postgres Database Context

The repository we built earlier requires some kind of database context in order to function. The database context is the core primitive of Entity Framework Core. To create a database context for the location model, we just need to create a class that inherits from DbContext. I've also included a **DbContextFactory because that can sometimes make running the Entity Framework Core command-line tools simpler:**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure; using
StatlerWaldorfCorp.LocationService.Models; using
Npgsql.EntityFrameworkCore.PostgreSQL;
namespace StatlerWaldorfCorp.LocationService.Persistence
{
public class LocationDbContext : DbContext
{
public LocationDbContext( DbContextOptions<LocationDbContext> options)
:base(options)
{
}
protected override void OnModelCreating( ModelBuilder modelBuilder)
{
base.OnModelCreating(modelBuilder);
modelBuilder.HasPostgresExtension("uuid-ossp");
}
```

```
public DbSet<LocationRecord> LocationRecords {get; set;}
}
public class LocationDbContextFactory :
IDbContextFactory<LocationDbContext>
{
public LocationDbContext Create(DbContextFactoryOptions options)
{
var optionsBuilder =
new DbContextOptionsBuilder<LocationDbContext>(); var connectionString =
Startup.Configuration
.GetSection("postgres:cstr").Value;
optionsBuilder.UseNpgsql(connectionString);

return new LocationDbContext(optionsBuilder.Options);
}
}
}
```

With a new database context, we need to make it available for dependency injection so that the location repository can utilize it:

```
public void ConfigureServices(IServiceCollection services)
{
var transient = true;
if (Configuration.GetSection("transient") != null) { transient =
Boolean.Parse(Configuration
.GetSection("transient").Value);
}
if (transient) { logger.LogInformation(
"Using transient location record repository.");
services.AddScoped<ILocationRecordRepository,
InMemoryLocationRecordRepository>();
} else {
var connectionString = Configuration.GetSection("postgres:cstr").Value;
services.AddEntityFrameworkNpgsql()
.AddDbContext<LocationDbContext>(options =>
options.UseNpgsql(connectionString));
logger.LogInformation(
"Using '{0}' for DB connection string.", connectionString);
services.AddScoped<ILocationRecordRepository, LocationRecordRepository>();
}
```

services.AddMvc();
}
The calls to AddEntityFrameworkNpgsql and AddDbContext are the magic that makes everything happen here.
With a context configured for DI, our service should be ready to run, test, and accept EF Core command-line parameters like the ones we need to execute migrations. When building your own database-backed services, you can also use the EF Core command-line tools to reverse-engineer migrations from existing database schemas.

Exercising the Data Service

Running the data service should be relatively easy. The first thing we're going to need to do is spin up a running instance of Postgres. If you were paying attention to the wercker.yml file for the location service that sets up the integration tests, then you might be able to guess at the docker run command to start Postgres with our preferred parameters:

```
$ docker run -p 5432:5432 --name some-postgres \
  -e POSTGRES_PASSWORD=inteword -e POSTGRES_USER=integrator \
  -e POSTGRES_DB=locationservice -d postgres
```

This starts the Postgres Docker image with the name some-postgres (this will be important shortly). To verify that we can connect to Postgres, we can run the following Docker command to launch psql:

```
$ docker run -it --rm --link some-postgres:postgres postgres \
  psql -h postgres -U integrator -d locationservice
```

At this point Postgres is running with a valid schema and it's ready to start accepting commands from the location service. Here's where it gets a little tricky. If we're going to run the location service from inside a Docker image, then referring to the Postgres server's host as localhost won't work —because that's the host inside the Docker image. What we need is for the location service to reach out of its container and then into the Postgres container. We can do this with a container link that creates a virtual hostname (we'll call it postgres), but we'll need to change our environment variable before launching the Docker image:

```
$ export POSTGRES__CSTR="Host=postgres;Username=integrator; \
Password=inteword;Database=locationservice;Port=5432"
$ docker run -p 5000:5000 --link some-postgres:postgres \
  -e TRANSIENT=false -e PORT=5000 \
  -e POSTGRES__CSTR dotnetcoreservices/locationservice:latest
```

Now that we've linked the service's container to the Postgres container via the postgres hostname, the location service should have no trouble connecting to the database. To see this all in action, let's submit a location record (as usual, take the line feeds out of this command when you type it):

```
$ curl -H "Content-Type:application/json" -X POST -d \
  '{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f","latitude":12.0, \
    "longitude":10.0,"altitude":5.0,"timestamp":0, \
    "memberId":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}' \
  http://localhost:5000/locations/63e7acf8-8fae-42ce-9349-3c8593ac8292
```

Take a look at the trace output from your running Docker image for the location service. You should see some very useful Entity Framework trace data explaining what happened. The service performed a SQL INSERT, so things are looking promising:

```
IRelationalCommandBuilderFactory[1]
      Executed DbCommand (23ms)
[Parameters=[@p0='?', @p1='?', @p2='?', @p3='?', @p4='?', @p5='?'],
 CommandType='Text', CommandTimeout='30']
      INSERT INTO "LocationRecords" ("ID", "Altitude", "Latitude",
"Longitude", "MemberID", "Timestamp")
      VALUES (@p0, @p1, @p2, @p3, @p4, @p5);
info: Microsoft.AspNetCore.Mvc.Internal.ObjectResultExecutor[1]
      Executing ObjectResult, writing value Microsoft.AspNetCore
.Mvc.ControllerContext.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action StatlerWaldorfCorp.LocationService.
Controllers.LocationRecordController.AddLocation
(StatlerWaldorfCorp.LocationService) in 2253.7616ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2602.7855ms 201 application/json;
charset=utf-8
```

Let's ask the service for this fictitious member's location history:

```
$ curl http://localhost:5000/locations/63e7acf8-8fae-42ce-9349-
3c8593ac8292

[{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f",
  "latitude":12.0,"longitude":10.0,"altitude":5.0,
  "timestamp":0,"memberID":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}]
```

The corresponding Entity Framework trace looks like this:

```
info: Microsoft.EntityFrameworkCore.Storage.
IRelationalCommandBuilderFactory[1]
      Executed DbCommand (23ms) [Parameters=[@__memberId_0='?'],
 CommandType='Text', CommandTimeout='30']
      SELECT "lr"."ID", "lr"."Altitude", "lr"."Latitude",
 "lr"."Longitude", "lr"."MemberID", "lr"."Timestamp"
      FROM "LocationRecords" AS "lr"
      WHERE "lr"."MemberID" = @__memberId_0
      ORDER BY "lr"."Timestamp"
```

Just to be double sure, let's query the latest endpoint to make sure we still get what we expect to see:

```
$ curl http://localhost:5000/locations/63e7acf8-8fae-42ce-9349-
3c8593ac8292 \
/latest

{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f",
  "latitude":12.0,"longitude":10.0,"altitude":5.0,
  "timestamp":0,"memberID":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}
```

Finally, to prove that we really are using real database persistence and that this isn't just a random fluke, use docker ps and docker kill to locate the Docker process for the location service and kill it. Restart it using the exact same command you used before. You should now be able to query the location service and get the exact same data you had before. Of course, once you stop the Postgres container you'll permanently lose that data.

PRACTICAL 8

Create an API gateway service

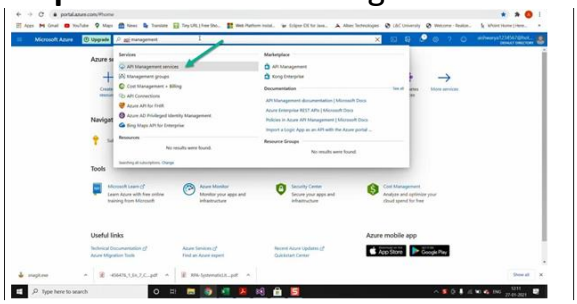
a) Create an API Management Service.

Steps for creating an API management service are mentioned below-

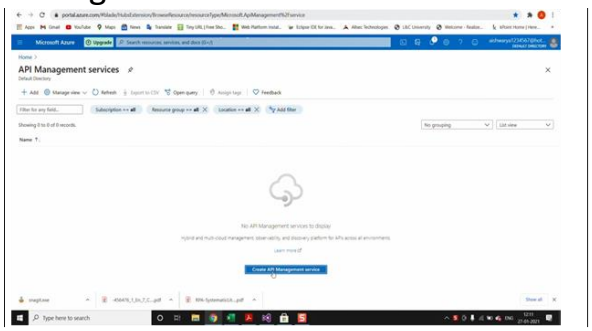
Step 1- Sign-in to your Azure Subscription Portal

Step 2- Search for “API Management”, then select API Management in order to create a service instance.

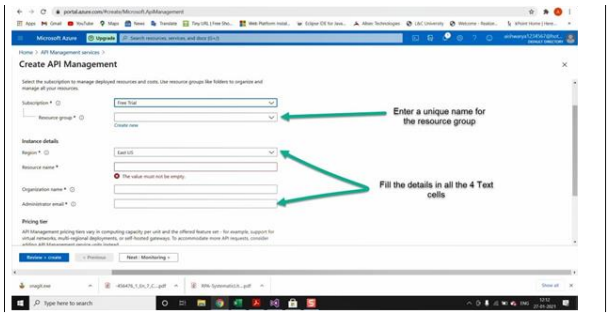
Step 3- As shown in Image 1 Select “API Management service”



Step 4- As API management service page is loaded, select “Create API management service” button. As shown in the image 2.



Step 5- We will be able to see the Azure API Management service creation blade as shown in Image after clicking on the “Create API management service” button



Step 6- Provide a name. This name sets the URL of the API gateway and portal [Name- AzureManagementService]

Step 7-Select the subscription and resource group (or create a new resource group), and select the location. [Resource Name- AzureManagementService, Location- SouthEast Asia]

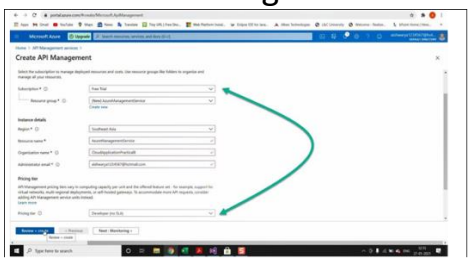
Step 8-Specify the organization name. This name will appear in the developer portal as the organization that publishes the API. [Name Cloud Application

Pratical

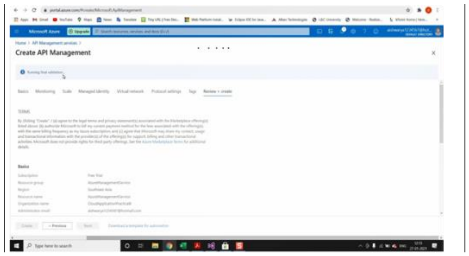
Step 9- Specify the email address of the administrator. The user who creates the service instance will be the default administrator, so it's best to provide the email address of this user until you want someone else to serve as administrator.

Step 10 -Select the pricing tier. The Developer tier is the most comprehensive offering, with sufficient request/response limitations in dev/test scenarios.

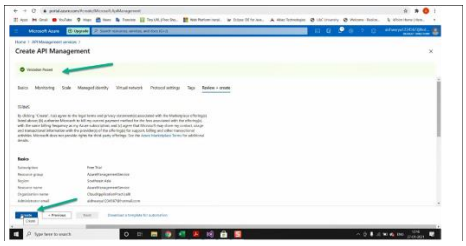
Step 11-After Completing the form click on review + create button to create “Azure API Management service instance”. As shown in image 4

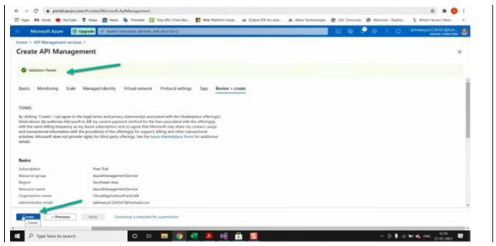


Step 12-As shown in the image 5 the status is “Running Final Validation”. Wait for it to be in “Validation Passed” Stage.

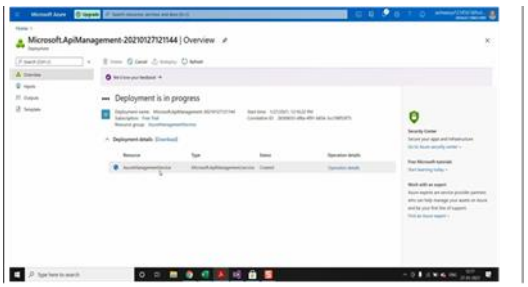


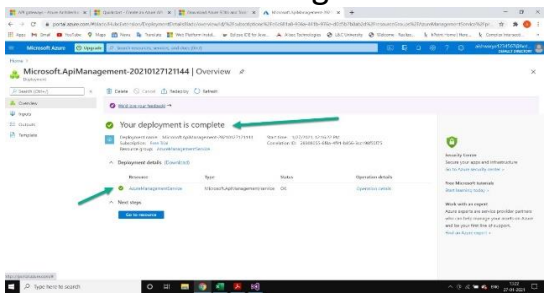
Step 13-Once the Validation is done and the label displays “Validation Passed”, Click on create button to create the instance of the service.





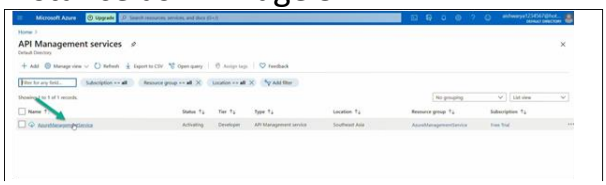
Step 14- After clicking on the create button the API management service instance will go to deployment stage as shown in image 7. After the deployment is complete the instance will be shown as in image 8



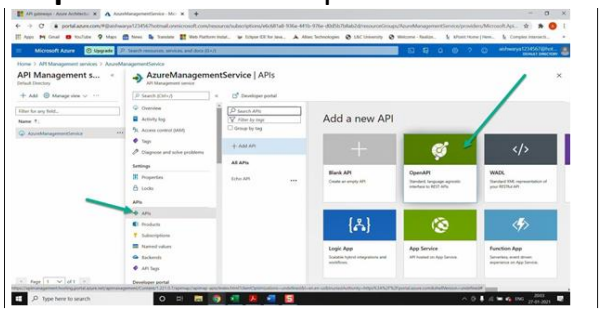


Next step is Import an API into API Management and test the API in the Azure portal

Step 15- In the Azure portal, search for and select API Management services.
Step 16- On the API Management services page, select your API Management instance as in image 9.



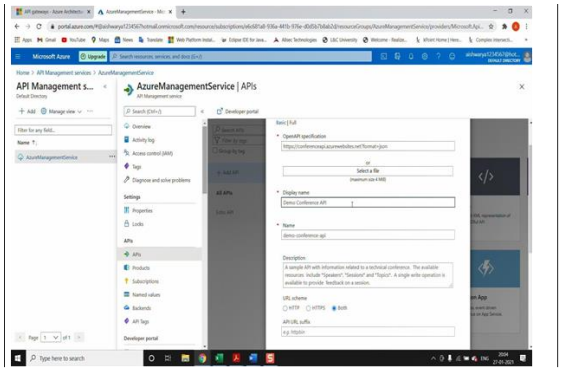
Step 17- In the left navigation of API Management instance, select APIs.
Step 18- Select the OpenAPI tile as shown in image 10.



Step 19- In the Create from OpenAPI specification window, select Full.
Step 20- Enter the values as mentioned below in the form shown in image 11

Setting	Value
OpenAPI specification	https://conferenceapi.azurewebsites.net?format=json
Display name	After you enter the preceding service URL, API Management fills out this field based on the JSON. .(automatic)

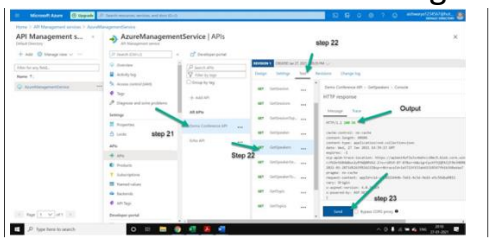
Name	After you enter the preceding service URL, API Management fills out this field based on the JSON. .(automatic)
Description	After you enter the preceding service URL, API Management fills out this field based on the JSON.(automatic)
URL scheme	HTTPS
API URL suffix	conference
Tags	-leave it as blank
Products	Unlimited
Gateways	Managed



Next steps are to Test the new API in the Azure portal

Step 21- In the left navigation of your API Management instance, select APIs > Demo Conference API.

Step 22- Select the Test tab, and then select GetSpeakers. The page shows Query parameters and Headers, if any. The Ocp-Apim-Subscription-Key is filled in automatically for the subscription key associated with this API. Step 22- Select Send as shown in image 12.

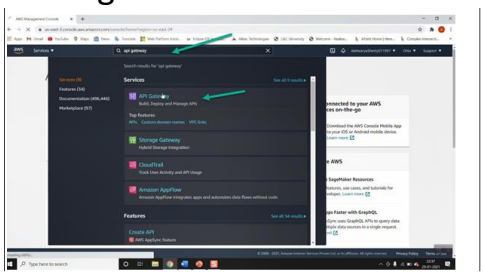


b) Create an API Gateway Service.

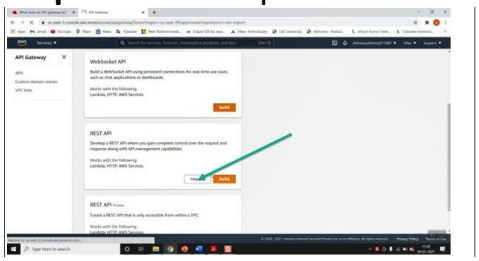
Steps for creating an API gateway service are mentioned below-

Step 1- Sign in to you AWS account.

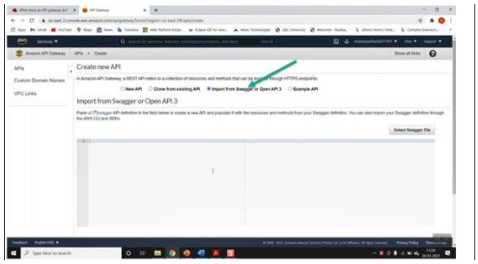
Step 2- search for API Gateway, and create an AWS Gateway instance as shown in Image 1.



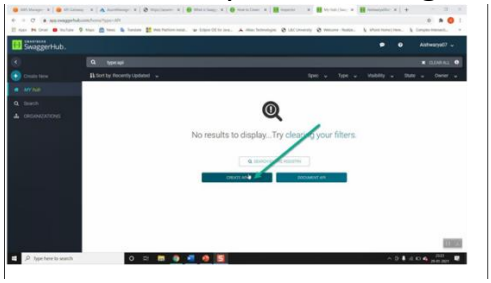
Step 3- Select Import from the REST API selection as shown in image 2.



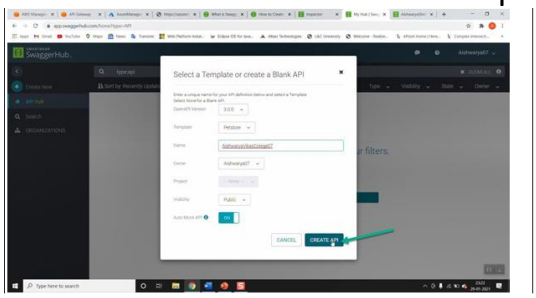
Step 4- Select Import from swagger and copy or select the swagger file as shown in image 3.



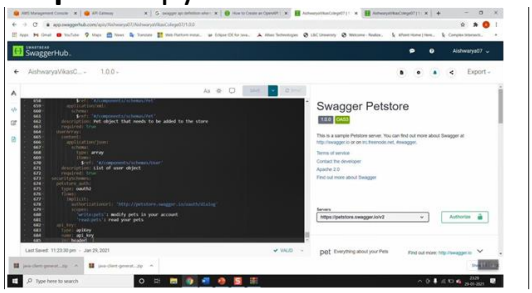
Step 5- To Create an API definition in Swagger. First login to the swagger account. As depicted in image 4.



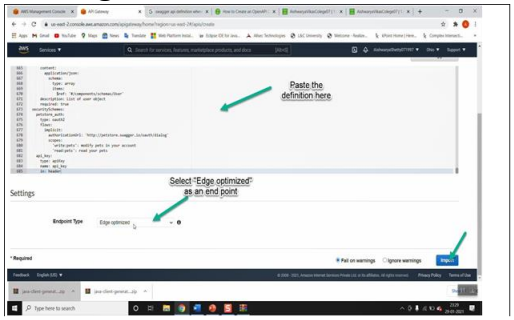
Step 6- Then fill the Form which includes the version, name and other details click on Create API button. As depicted in image 5.



Step 7- Copy the API and definition from swagger as shown in image 6



Step 8-Paste the definition and in In the Settings section, select the endpoint type. Select “Edge Optimized” option and then click on import button as shown in image 7



API definition from swagger is mentioned below- openapi: 3.0.0

```
info:
description: |
This is a sample Petstore server. You can find out more about Swagger at
[http://swagger.io](http://swagger.io) or on [irc.freenode.net,
#swagger](http://swagger.io/irc/). version: "1.0.0"
title: Swagger Petstore
termsOfService: 'http://swagger.io/terms/' contact:
email: apiteam@swagger.io license:
name: Apache 2.0
url: 'http://www.apache.org/licenses/LICENSE-2.0.html' servers:
# Added by API Auto Mocking Plugin
-       description: SwaggerHub API Auto Mocking
url:
https://virtserver.swaggerhub.com/Aishwarya07/AishwaryaVikasColege07/1
.0.0
-       url: 'https://petstore.swagger.io/v2' tags:
-       name: pet
description: Everything about your Pets externalDocs:
description: Find out more url: 'http://swagger.io'
-       name: store
description: Access to Petstore orders
-       name: user
description: Operations about user externalDocs:
description: Find out more about our store url: 'http://swagger.io'
paths:
/pet:
post:
tags:
- pet
summary: Add a new pet to the store operationId: addPet
responses:
'405':
description: Invalid input security:
-       petstore_auth:
-       'write:pets'
-       'read:pets' requestBody:
$ref: '#/components/requestBodies/Pet' put:
tags:
-       pet
summary: Update an existing pet operationId: updatePet
```

responses:
'400':
description: Invalid ID supplied '404':
description: Pet not found '405':
description: Validation exception security:
- petstore_auth:
- 'write:pets'
- 'read:pets' requestBody:
$ref: '#/components/requestBodies/Pet'
/pet/findByStatus:
get:
tags:
- pet
summary: Finds Pets by status
description: Multiple status values can be provided with comma separated strings
operationId: findPetsByStatus parameters:
- name: status in: query
description: Status values that need to be considered for filter required: true
explode: true schema:
type: array items:
type: string enum:
- available
- pending
- sold
default: available responses:
'200':
description: successful operation content:

application/json:
schema:
type: array items:
$ref: '#/components/schemas/Pet' application/xml:
schema:
type: array items:
$ref: '#/components/schemas/Pet' '400':
description: Invalid status value security:
- petstore_auth:
- 'write:pets'
- 'read:pets'

/pet/findByTags:
get:
tags:
- pet
summary: Finds Pets by tags description: >-
Muliple tags can be provided with comma separated strings. Use\ \ tag1, tag2, tag3 for testing.
operationId: findPetsByTags parameters:
- name: tags in: query
description: Tags to filter by required: true
explode: true schema:
type: array items:
type: string responses:
'200':
description: successful operation content:

application/json:
schema:
type: array items:
$ref: '#/components/schemas/Pet' application/xml:
schema:
type: array items:
$ref: '#/components/schemas/Pet' '400':
description: Invalid tag value security:
- petstore_auth:
- 'write:pets'
- 'read:pets' deprecated: true
'/pet/{petId}':
get:
tags:
- pet
summary: Find pet by ID
description: Returns a single pet operationId: getPetById
parameters:
- name: petId in: path
description: ID of pet to return required: true
schema:
type: integer format: int64
responses:
'200':

description: successful operation content:
application/json:
schema:
$ref: '#/components/schemas/Pet'

application/xml:
schema:
$ref: '#/components/schemas/Pet' '400':
description: Invalid ID supplied '404':
description: Pet not found security:
- api_key: [] post:
tags:
- pet
summary: Updates a pet in the store with form data operationId:
updatePetWithForm
parameters:
- name: petId in: path
description: ID of pet that needs to be updated required: true
schema:
type: integer format: int64
responses:
'405':
description: Invalid input security:
- petstore_auth:
- 'write:pets'
- 'read:pets' requestBody:
content:
application/x-www-form-urlencoded: schema:
type: object properties:
name:
description: Updated name of the pet type: string
status:

delete: tags:
- pet

description: Updated status of the pet type: string

summary: Deletes a pet operationId: deletePet parameters:
- name: api_key in: header
required: false schema:
type: string
- name: petId in: path
description: Pet id to delete required: true
schema:
type: integer format: int64
responses:
'400':
description: Invalid ID supplied '404':
description: Pet not found security:
- petstore_auth:
- 'write:pets'
- 'read:pets'
'/pet/{petId}/uploadImage': post:
tags:
- pet
summary: uploads an image operationId: uploadFile parameters:
- name: petId in: path
description: ID of pet to update

required: true schema:
type: integer format: int64
responses:
'200':
description: successful operation content:
application/json:
schema:
$ref: '#/components/schemas/ApiResponse' security:
- petstore_auth:
- 'write:pets'
- 'read:pets' requestBody:
content:
application/octet-stream:
schema:
type: string
format: binary
/store/inventory:
get:

tags:
- store
summary: Returns pet inventories by status
description: Returns a map of status codes to quantities operationId:
getInventory
responses:
'200':
description: successful operation content:
application/json: schema:
type: object
additionalProperties: type: integer
format: int32 security:

- api_key: []
/store/order:
post:
tags:
- store
summary: Place an order for a pet operationId: placeOrder
responses:
'200':
description: successful operation content:
application/json:
schema:
$ref: '#/components/schemas/Order' application/xml:
schema:
$ref: '#/components/schemas/Order' '400':
description: Invalid Order requestBody:
content:
application/json:
schema:
$ref: '#/components/schemas/Order'
description: order placed for purchasing the pet required: true
'/store/order/{orderId}':
get:
tags:
- store
summary: Find purchase order by ID description: >-
For valid response try integer IDs with value >= 1 and <= 10.\ \ Other values
will generated exceptions

operationId: getOrderById parameters:
- name: orderId in: path
description: ID of pet that needs to be fetched

required: true schema:
type: integer format: int64 minimum: 1
maximum: 10 responses:
'200':
description: successful operation content:
application/json:
schema:
$ref: '#/components/schemas/Order' application/xml:
schema:
$ref: '#/components/schemas/Order' '400':
description: Invalid ID supplied '404':
description: Order not found delete:
tags:
- store
summary: Delete purchase order by ID description: >-
For valid response try integer IDs with positive integer value.\ \ Negative or non-integer values will generate API errors
operationId: deleteOrder parameters:
- name: orderId in: path
description: ID of the order that needs to be deleted required: true
schema: type: integer
format: int64 minimum: 1
responses: '400':

description: Invalid ID supplied '404':
description: Order not found
/user:
post:
tags:
- user
summary: Create user
description: This can only be done by the logged in user. operationId:
createUser
responses:
default:
description: successful operation requestBody:

content:
application/json:
schema:
$ref: '#/components/schemas/User' description: Created user object
required: true
/user/createWithArray:
post:
tags:
- user
summary: Creates list of users with given input array operationId:
createUsersWithArrayInput
responses:
default:
description: successful operation requestBody:
$ref: '#/components/requestBodies/UserArray'
/user/createWithList:
post:
tags:
- user
summary: Creates list of users with given input array operationId:
createUsersWithListInput
responses: default:

description: successful operation requestBody:
$ref: '#/components/requestBodies/UserArray'
/user/login:
get:
tags:
- user
summary: Logs user into the system operationId: loginUser
parameters:
- name: username in: query
description: The user name for login required: true
schema:
type: string
- name: password in: query
description: The password for login in clear text required: true
schema:
type: string responses:
'200':

description: successful operation headers:
X-Rate-Limit:
description: calls per hour allowed by the user schema:
type: integer format: int32
X-Expires-After:
description: date in UTC when token expires schema:
type: string
format: date-time content:
application/json: schema:

type: string application/xml:
schema:
type: string '400':
description: Invalid username/password supplied
/user/logout:
get:
tags:
- user
summary: Logs out current logged in user session operationId: logoutUser
responses:
default:
description: successful operation '/user/{username}':
get:
tags:
- user
summary: Get user by user name operationId: getUserByName parameters:
- name: username in: path
description: The name that needs to be fetched. Use user1 for testing.
required: true
schema:
type: string responses:
'200':
description: successful operation content:
application/json:
schema:
$ref: '#/components/schemas/User' application/xml:
schema:
$ref: '#/components/schemas/User' '400':

description: Invalid username supplied '404':

description: User not found put:
tags:
- user
summary: Updated user
description: This can only be done by the logged in user. operationId:
updateUser
parameters:
- name: username in: path
description: name that need to be updated required: true
schema:
type: string responses:
'400':
description: Invalid user supplied '404':
description: User not found requestBody:
content:
application/json:
schema:
$ref: '#/components/schemas/User' description: Updated user object
required: true delete:
tags:
- user
summary: Delete user
description: This can only be done by the logged in user. operationId:
deleteUser
parameters:
- name: username in: path
description: The name that needs to be deleted required: true

schema:
type: string responses:
'400':
description: Invalid username supplied '404':
description: User not found externalDocs:
description: Find out more about Swagger url: 'http://swagger.io'
components:
schemas:
Order:
type: object properties:
id:
type: integer format: int64 petId:

type: integer format: int64 quantity:
type: integer format: int32 shipDate:
type: string
format: date-time status:
type: string
description: Order Status enum:
- placed
- approved
- delivered complete:
type: boolean default: false
xml:
name: Order
Category:
type: object properties:
id:
type: integer format: int64 name:
type: string xml:
name: Category User:
type: object properties:
id:
type: integer format: int64 username:
type: string firstName:
type: string lastName:
type: string email:
type: string password:
type: string phone:
type: string userStatus:
type: integer format: int32
description: User Status xml:
name: User Tag:
type: object properties: id:
type: integer

format: int64 name:
type: string xml:
name: Tag Pet:
type: object required:
- name
- photoUrls properties:
id:

type: integer format: int64 category:
$ref: '#/components/schemas/Category' name:
type: string example: doggie photoUrls:
type: array xml:
name: photoUrl wrapped: true items:
type: string tags:
type: array xml:
name: tag wrapped: true items:
$ref: '#/components/schemas/Tag' status:
type: string
description: pet status in the store enum:
- available
- pending

- sold xml:
name: Pet ApiResponse:
type: object properties:
code:
type: integer format: int32 type:
type: string message:
type: string requestBodies:
Pet:
content:
application/json:
schema:
$ref: '#/components/schemas/Pet' application/xml:
schema:
$ref: '#/components/schemas/Pet'
description: Pet object that needs to be added to the store required: true
UserArray:
content:
application/json:
schema:
type: array items:
$ref: '#/components/schemas/User' description: List of user object
required: true securitySchemes:
petstore_auth:
type: oauth2 flows:
implicit:
authorizationUrl: 'http://petstore.swagger.io/oauth/dialog'

scopes:
'write:pets': modify pets in your account 'read:pets': read your pets
api_key:
type: apiKey name: api_key in: header
Output-
If the import is successful, we can view the “API Structure in AWS API Gateway” where we can see methods of the API. As shown in image 8