

Contents

Introduction	1
Pre-requisites	1
Setup Microservice Extractor	2
Review Sample application	3
Onboard the sample application to AWS Microservice Extractor	5
Launch Visualization	6
Create a group for extraction.	8
Extract a microservice	10
Post Extraction	10
Conclusion	12

Introduction

This guide shows a step-by-step process to extract a Microservice from an ASP.NET 4.7 based monolithic application that uses MVC 5 and Entity Framework 6.0. The workshop uses a demo online shopping application with AWS Microservice Extractor to create a new Microservice and update existing monolith code to make remote API calls to the extracted Microservice.

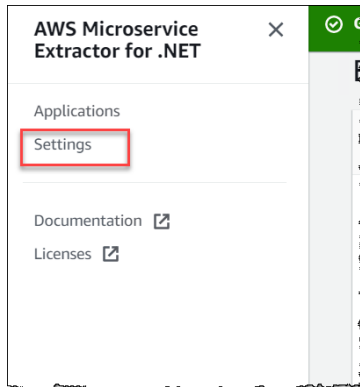
Pre-requisites

If you are doing this workshop on a machine you control (such as your laptop or a Virtual machine), you will need to ensure that you have following components installed.

1. Visual Studio 2019 with the below features enabled
 1. Under “Workloads”
 1. “ASP.NET and Web Development”
 2. Under Individual Components
 1. .NET 4.7.1 Targeting Pack
 2. SQL Server Express 2019.
 3. SQL Server Management Studio.
2. An AWS IAM user profile already setup on your machine or access to IAM user’s access key ID and secret access key to configure a new profile for the user. You can create a new IAM user using [these instructions](#).
3. Download and Install AWS Microservice extractor for .NET from [here](#).
4. Download and Install Git client from [here](#).

Setup Microservice Extractor

1. Launch Microservice Extractor and click on settings link in the left pane



2. In the Setup details page;
 1. Provide **AWS Region** you are working in.
 2. Select **AWS Named profile** if it already exists. If you don't see a profile click on **Add a named profile** to create a new profile. You will need an IAM user's access key ID and secret access key to create a profile.
 3. Microservice Extractor uses **Working directory** to store extracted code and other artifacts. Provide a folder location which can be used as a working directory.
 4. Microservice extractor uses MSBuild to build application project. **MSBuild path** is automatically detected if you have Visual Studio 2019 installed on your machine.
 5. Check the **usage data sharing** checkbox.

A screenshot of the 'Set up details' page in the AWS Microservice Extractor for .NET application. The page has a header with the title 'Set up details' and an 'Info' link. The main content area is divided into several sections:

- Region**: A dropdown menu with 'us-west-2' selected.
- AWS named profile**: A dropdown menu with 'Runeetv@isengard' selected. Below it is a link 'Add a named profile'.
- Working directory**: A text field with 'C:\Drift-Working-Dir' and a green checkmark icon. Above it is a button 'Update directory'.
- MSBuild path**: A text field with 'C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\MSBuild\Current\Bin\MSBuild.exe' and a green checkmark icon. Above it is a button 'Update Path'.
- AWS Microservice Extractor for .NET usage data sharing**: A section with a description and a checkbox 'I agree to share my usage data with AWS Microservice Extractor for .NET' which is checked. Below it is a note: 'Usage data will be shared to the us-east-1 region regardless of selected region above.'

Review Sample application

1. Download or clone sample ASP.NET MVC application using following command

Clone command:

git clone <https://github.com/runeetv/GadgetsOnline-DotNet.git>

Download location:

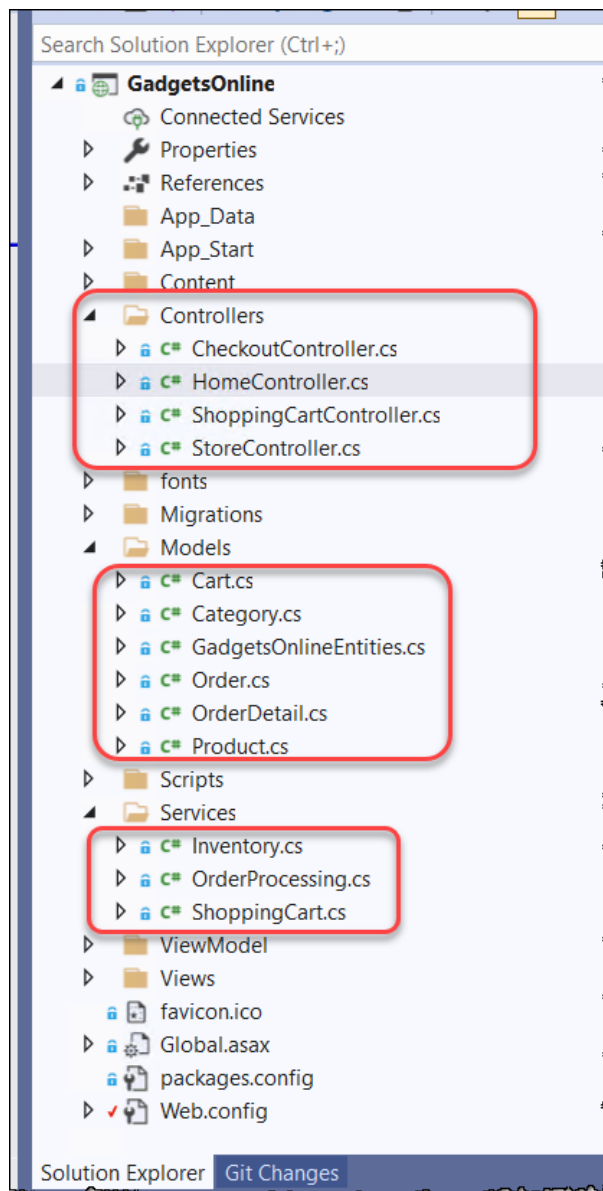
<https://github.com/runeetv/GadgetsOnline-DotNet>

2. Unzip and restore DB backup from **GadgetsOnlineDB_backup.zip** to your local SQL Server instance.
3. Open downloaded sample application in Visual studio by double clicking **GadgetsOnline.sln**.
4. Double click web.config and update connectionString **GadgetsOnlineEntities** to point to correct SQL Server instance.
5. Launch sample application in debug mode by pressing F5 button or by going to **Debug >> Start Debugging**.
6. You should be able to see the Home page of a sample online shopping web app. The left navigation bar shows the list of product categories, and the bottom of the page shows best-selling products. Clicking on products shows product information. Clicking on categories shows products in that category.

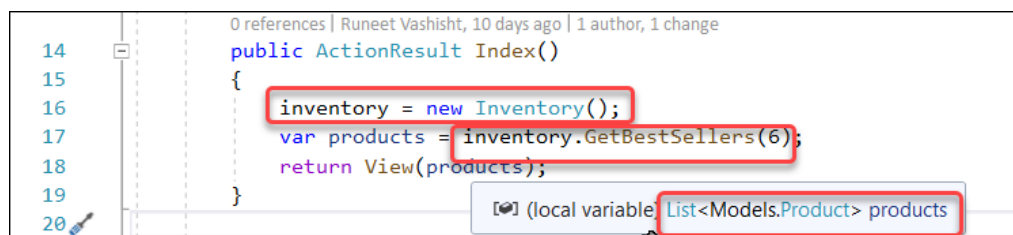


- Let's look at the structure of the project in Visual Studio. MVC **Controllers** accept user requests and create an instance of the business **Service**. **Service** class creates an instance of **GadgetsOnlineEntities** DbContext and returns one or more DB models (**Products, Orders, Category** etc) to the calling Controller.

The application simulates a three-tier app where Controllers call business services to complete user requests and the services in turn call database access layer. This is a very common pattern used in ASP.NET MVC applications.



- Review **Index** action in **HomeController.cs**



- On Line 16, we are creating an instance of Inventory class.

- On Line 17, we are calling **GetBestSellers()** method to get top 6 selling products to be listed on the home page.
 - The **GetBestSellers()** method returns List of Product data model.
9. Go ahead and review **Inventory.cs** class under Services folder. You will see multiple methods using an instance of **GadgetsOnlineEntities** DbContext to fetch data models and filtering based on one or more conditions.

Onboard the sample application to AWS Microservice Extractor.

Now that you understand the application structure. Let's onboard GadgetsOnline sample application to AWS Microservice Extractor.

1. Go back to Microservice Extractor and click on **Applications** link on top left.
2. In the right pane, click on **"Onboard application"**.
3. In the **Application details** page;
 - Provide Name as **"GadgetsOnline"**
 - Under **Source Code** provide the location of the sample application's solution (.sln) file.
 - **Runtime profiling data** includes runtime usage metrics which can be overlaid on the visualization graph in next steps. This is optional and we can skip this setting for this lab.
 - **Analyze .NET Core Portability** is also an optional setting. Enabling this setting requires [Porting Assistant for .NET](#) to be installed on same machine as Microservice Extractor. This setting integrates source code analysis with Porting Assistant to find the code compatibility with .NET Core. We can skip this setting for this lab.
4. Click **"Onboard application"**. This will start the source code analysis by Microservice Extractor.
 - This analysis will be used to create a directed dependency graph between various classes of the application.
 - This process can take couple of minutes. Once completed, you are ready to visualize dependency graph created by Microservice Extractor.

Launch Visualization

Once onboarding status shows **“Success”**, you can launch visualization either by clicking **“View dependency graph”** on the green banner at top or by selecting the application from the Applications list and clicking on **“Launch visualization”** button.

The screenshot shows the AWS Microservice Extractor for .NET interface. At the top, a green banner displays a checkmark icon and the text "GadgetsOnline is ready for visualization". To the right of this banner is a button labeled "View dependency graph". Below the banner, the breadcrumb "AWS Microservice Extractor for .NET > Applications" is visible. The main content area is titled "Get started" and contains a section for "AWS Microservice Extractor for .NET" with a description and a four-step process: 1. Onboard application, 2. Launch visualization, 3. Refactor and extract, and 4. Build and deploy. Below this, the "Applications (1)" section shows a table of applications. The table has columns for Name, Application type, Source code location, Onboarding status, and Date created. The first application, "GadgetsOnline", has an onboarding status of "Success". A red box highlights the "Onboarding status" column header and the "Success" status for the "GadgetsOnline" application. A red arrow points from the "Launch visualization" button to the "Onboarding status" column header. Another red arrow points from the "View dependency graph" button to the "Launch visualization" button.

Get started

AWS Microservice Extractor for .NET

AWS Microservice Extractor for .NET identifies parts of your application to extract as new services after analyzing the source code and runtime metrics.

- 1 Onboard application**
Onboard your application by providing access to buildable source code, and optionally running AWS Microservice Extractor for .NET's profiler to collect runtime metrics.
- 2 Launch visualization**
Use AWS Microservice Extractor for .NET's analysis and visualization tools to group and label parts of the application to create as independent services.
- 3 Refactor and extract**
Refactor source code by isolating business domains and removing dependencies between them. Extract groups as separate code repositories.
- 4 Build and deploy**
Manually refactor, build and deploy the extracted services as well as the modified application code.

Applications (1)
.NET web applications that contain buildable source code and its dependencies.

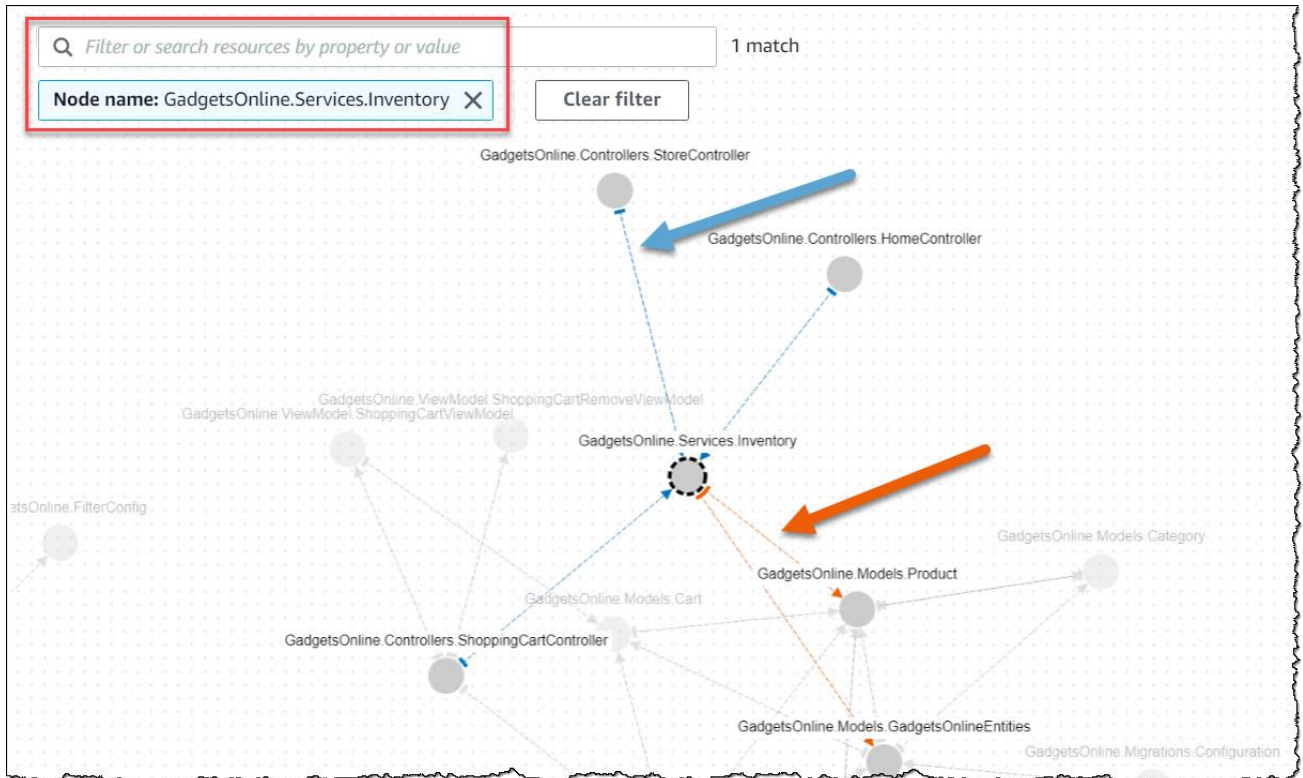
Launch visualization Action Onboard application

Filter applications by name

Name	Application type	Source code location	Onboarding status	Date created
GadgetsOnline	.NET	C:\repos\lab\GadgetsOnline-DotNet\GadgetsOnline\GadgetsOnline.sln	Success	12/8/2021 5:06:41 AM

Visualization tab shows a directed dependency graph where every node in the graph denotes a class and an edge between the nodes denote dependency between the nodes.

1. In the search/filter box search for Inventory to list **GadgetsOnline.Services.Inventory** class. Scroll the mouse to zoom in on the highlighted section.



2. The blue edges show incoming dependency. In this case MVC Controllers **Store, Home, and ShoppingCart** are dependent on an instance of an **Inventory** class.
3. The orange edges show outbound dependency from a node. In this case **Inventory** class has direct dependency on **GadgetsOnline** DbContext and data models such as **Products**.

Above visualization and dependency graphs illustrates that if we extract **Inventory** as a separate microservice, we need to re-factor three controllers (**Store, ShoppingCart and Home**) to make remote API calls instead of using an instance of local **Inventory** class. We will see in upcoming steps that Microservice Extractor will be able help us with this refactoring.

Additionally, if we extract **Inventory** as a separate microservice, we will have to copy of the dependencies of **Inventory** class in the new microservice. We will see in upcoming steps that Microservice Extractor will help us with copying dependencies in a new ASP.NET Web API project for extracted microservice.

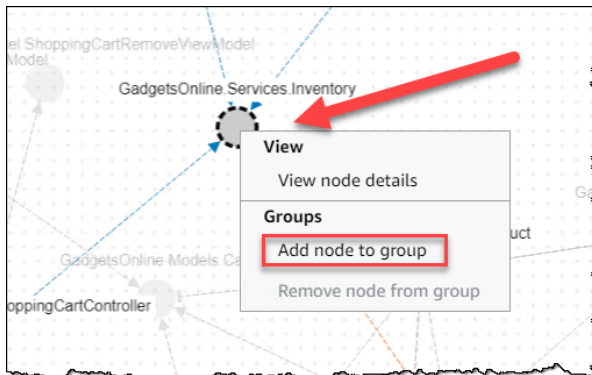
Before we go to next step and extract **Inventory** as a separate Microservice, it is worth reviewing various visualization options available with Microservice extractor. Review [this blog](#) to see **Namespace view** and **Island view** as alternate visualizations supported by Microservice extractor.

Create a group for extraction.

Once you visualize dependencies between various parts of your application, you can begin extracting the process. The extraction process requires organizing classes as a group which can be extracted as an independent service. This step might require refactoring your existing code before extraction.

In this example, you will extract Inventory as an independent service.

1. Click “Clear filter” button to clear existing filters.
2. Right click on **Inventory** node and click **Add node to group**.



3. Select **Create new** to create a new group.
Give **Group name** as **Inventory**.
Choose a **Group Color** from the drop down and click **Add** button.

Add node to group

Selected node (1)

Class ID	Associated group
GadgetsOnline.Services.Inventory	<input type="radio"/> No group

Add to group

Group type

Select to add the node to an existing group or create a new group.

☐ Existing ☒ Create new

Group name

Enter a name for the group.

Inventory

Maximum of 20 characters. Letters, numbers, spaces, '-', and '_' are allowed.

Group color - optional

Select a color to apply to nodes in this group.

Cancel

Add

4. You will see a new group created. Click on the Inventory group to see **Group details**.

Note the list of the **Nodes** show **Inventory** class. This class will be extracted as an independent service and it will be exposed as a REST API post extraction. Review the **Dependencies** list, all the dependencies in this list will be copied or referenced in the extracted independent service.



The screenshot shows the 'Group details' dialog box with the 'General' tab selected. The 'Node (1)' section shows a single node: 'clr:GadgetsOnline.Services.Inventory'. A red arrow points to the 'Extract group' button. The 'Dependencies (20)' section shows a list of dependencies, including 'clr:GadgetsOnline.Models.Cart', 'clr:GadgetsOnline.Models.Category', 'clr:GadgetsOnline.Models.GadgetsOnlineEntities', and 'clr:GadgetsOnline.Models.Order'.

Class ID	Comments
clr:GadgetsOnline.Services.Inventory	-

Class ID	Comments
clr:GadgetsOnline.Models.Cart	-
clr:GadgetsOnline.Models.Category	-
clr:GadgetsOnline.Models.GadgetsOnlineEntities	-
clr:GadgetsOnline.Models.Order	-

5. Click on **Extract group** button to begin extraction process.

Extract a microservice

Follow these steps on **Review details and initiate extraction** screen

1. Provide service name as **InventoryService**
2. Review **Nodes** to be extracted. This should list **Inventory** class from the sample application.
Review **Dependencies** list, these dependencies will be copied or referenced in the extracted independent service.
3. In the **Method invocation from the application** section, select **Use remote method invocations**. This setting will allow Microservice extractor to replace local Inventory class invocations with the remote REST API calls as part of the extraction process.

Method invocations from the application to the extracted service

When a group from the source application is extracted as a service, AWS Microservice Extractor for .NET creates two new code repositories:

1. **The extracted service** contains all of the dependencies of the group being extracted. AWS Microservice Extractor for .NET copies the classes and dependencies to the extracted service.
2. **The modified application** is the copy of the original application with modifications for the isolated service.

Choose how you want AWS Microservice Extractor for .NET to handle method calls to the extracted service:

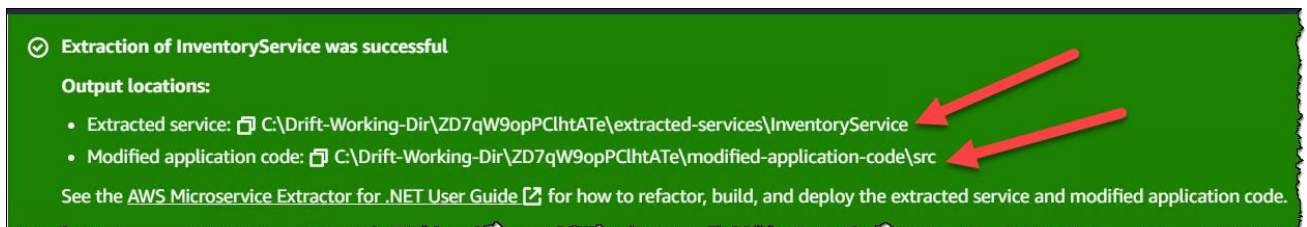
☒ **Use remote method invocations**
AWS Microservice Extractor for .NET will replace the local method invocations with remote invocations to the extracted service, where possible. Network calls can add additional latency to user requests.

☐ **Use local method invocations**
The modified application will continue to invoke the local methods. Comments will be added to these invocations to help you refactor the code to call the extracted service.

4. Click on the **Extract** button.

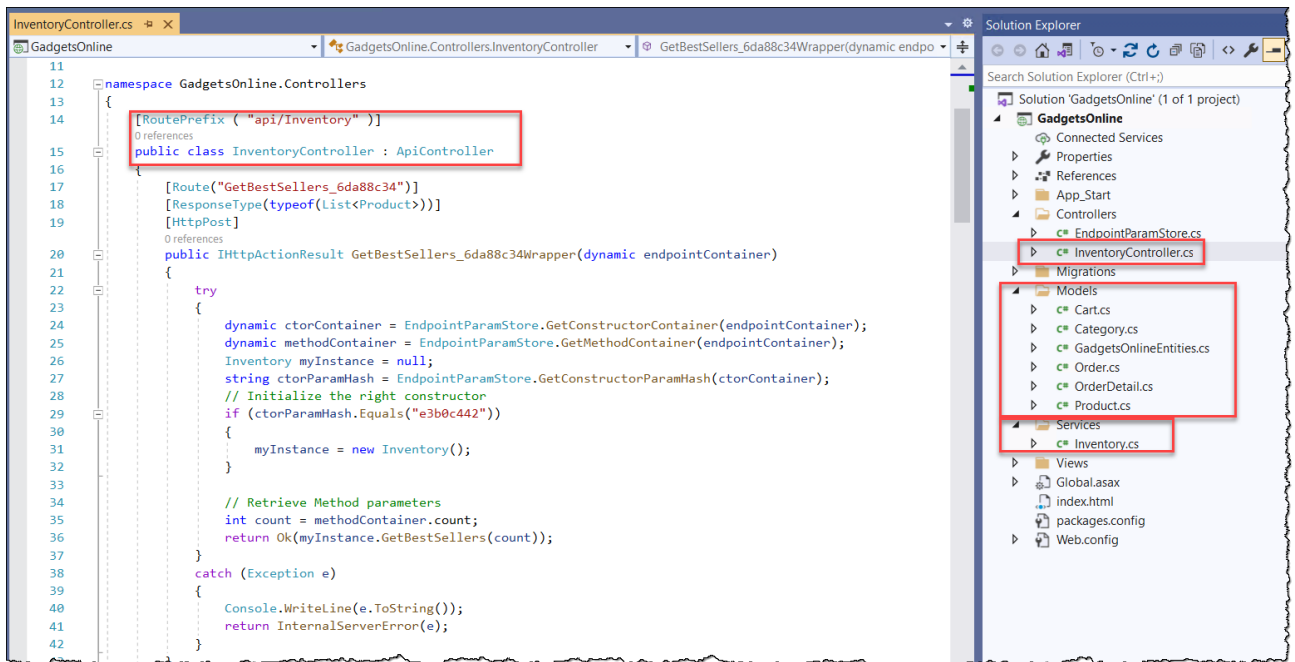
Post Extraction

1. Once extraction is completed. You can copy the **Output locations** of the **Extracted service** and the **Modified application code**.

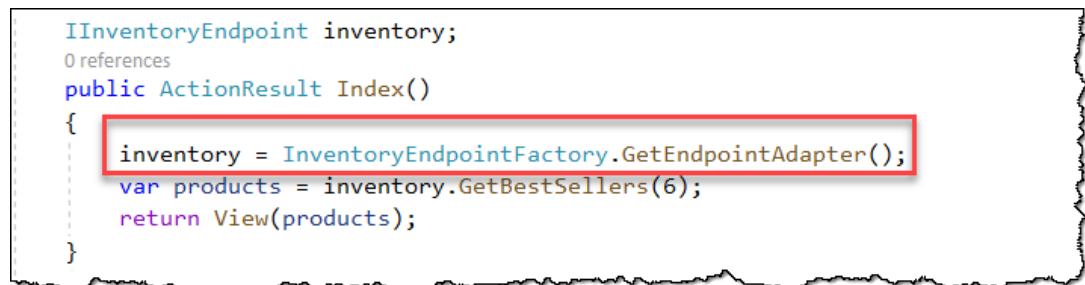


2. Double click the solution file in **Extracted service** folder to open the ASP.NET WebAPI project in visual studio and review the project structure.
 - You will find **Inventory** class copied under **Services** folder.
 - You will see all the depending entity Framework data models and DB context copied under Models folder.
 - You will see an additional **ApiController** called **InventoryController** created. This controller has one to one mapping with Inventory class exposing public methods as REST APIs.

Note that Microservice Extractor will make best efforts to making the newly created service compile able but it is not guaranteed. You may still have to fix references or update Nuget packages to compile the service.



3. Double click the solution file in **Modified application code** folder to open the copy of the updated monolith code.
 1. Review **HomeController**, **ShoppingCartController** and **StoreController**. Local Inventory class calls are converted to remote API calls using **EndpointAdapter**.



2. Microservice extractor used heuristics to change local class calls into remote API calls. You may have to make this change manually or may have to refactor the changes made by the Extractor.
3. Review the **EndpointAdapter** folder. It has an **InventoryEndpointFactory** class which can switch between Remote REST API calls and local Inventory class calls based on a flag called **RemoteRoutingToMicroservice**.



Conclusion

We saw that AWS Microservice Extractor for .NET can assist you in extracting microservices from your .NET monolithic applications. By using this tool, you can simplify, and accelerate your migration from monolithic applications to microservices.