AWS Microservice Extractor for .NET workshop.

Contents

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.

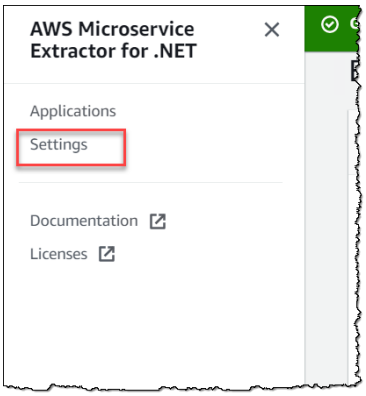
Pre-requisites

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.

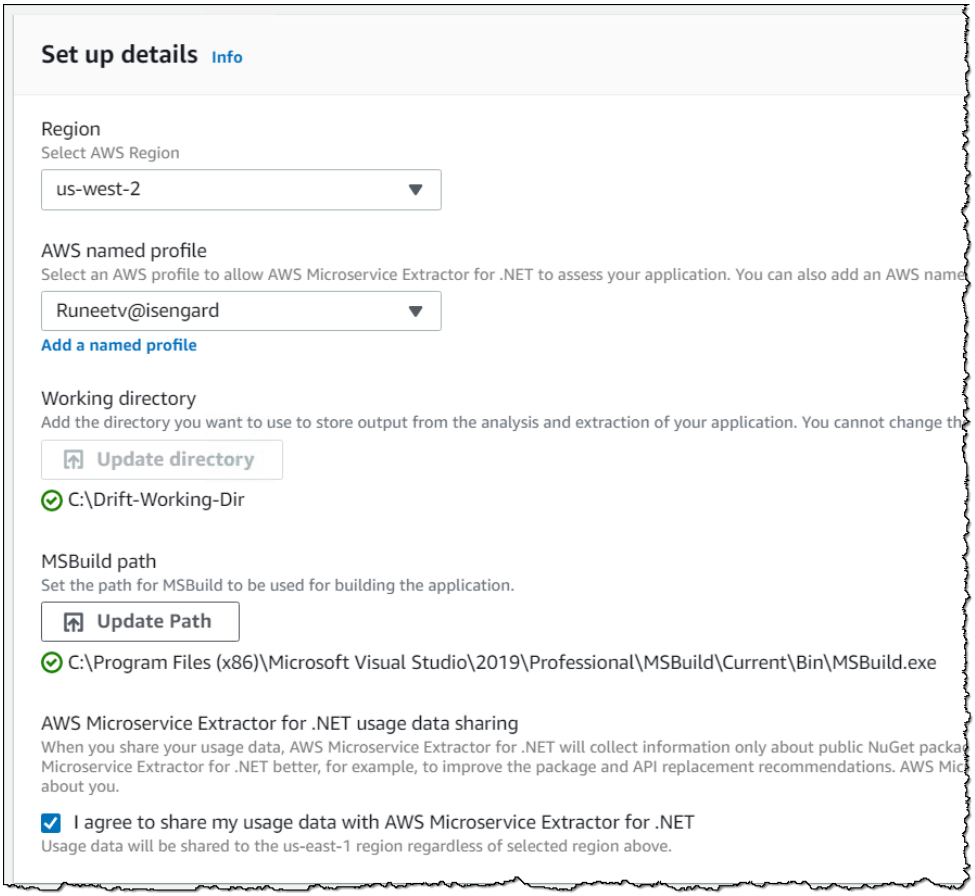
1. Visual Studio 2019 with the below features enabled
    1. Under “Workloads”
        1. “ASP.NET and Web Development”
    2. Under Individual Components
        1. .NET 4.7.1 Targeting Pack
        2. SQL Server Express 2019.
        3. SQL Server Management Studio.
2. An AWS IAM user profile already 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.



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.**

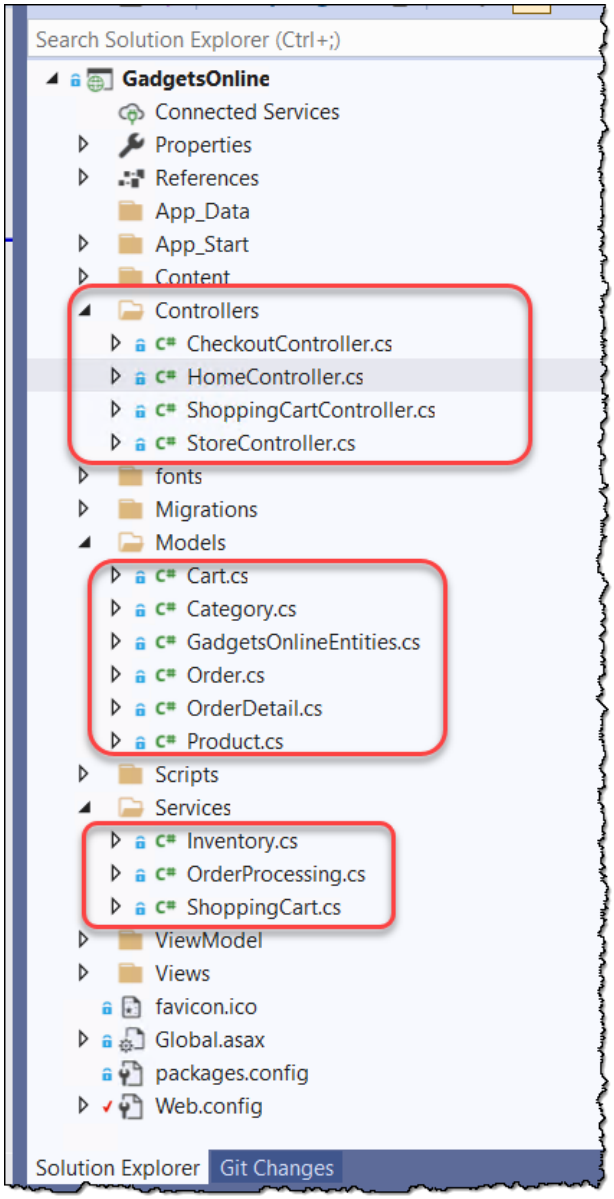
4. Double click web.config and update connectionString **GadgetsOnlineEntities** to point to correct SQL Server instance.

5. Launch sample application in debug mode by pressing F5 button or by going to **Debug** >> **Start Debugging**.

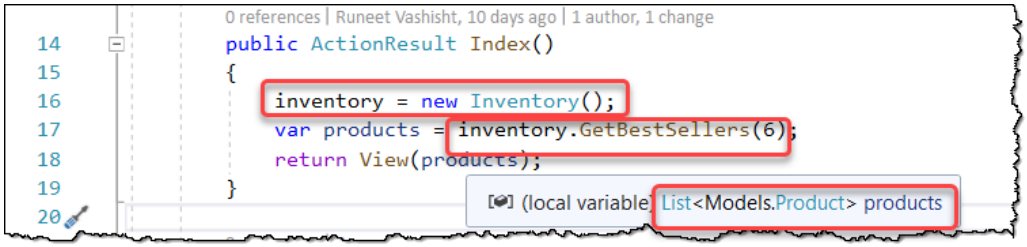
6. You should be able to see the Home page of a sample online shopping web app. The left navigation bar shows the list of product categories, and the bottom of the page shows best-selling products. Clicking on products shows product information. Clicking on categories shows products in that category.

7. 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.



8. 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.

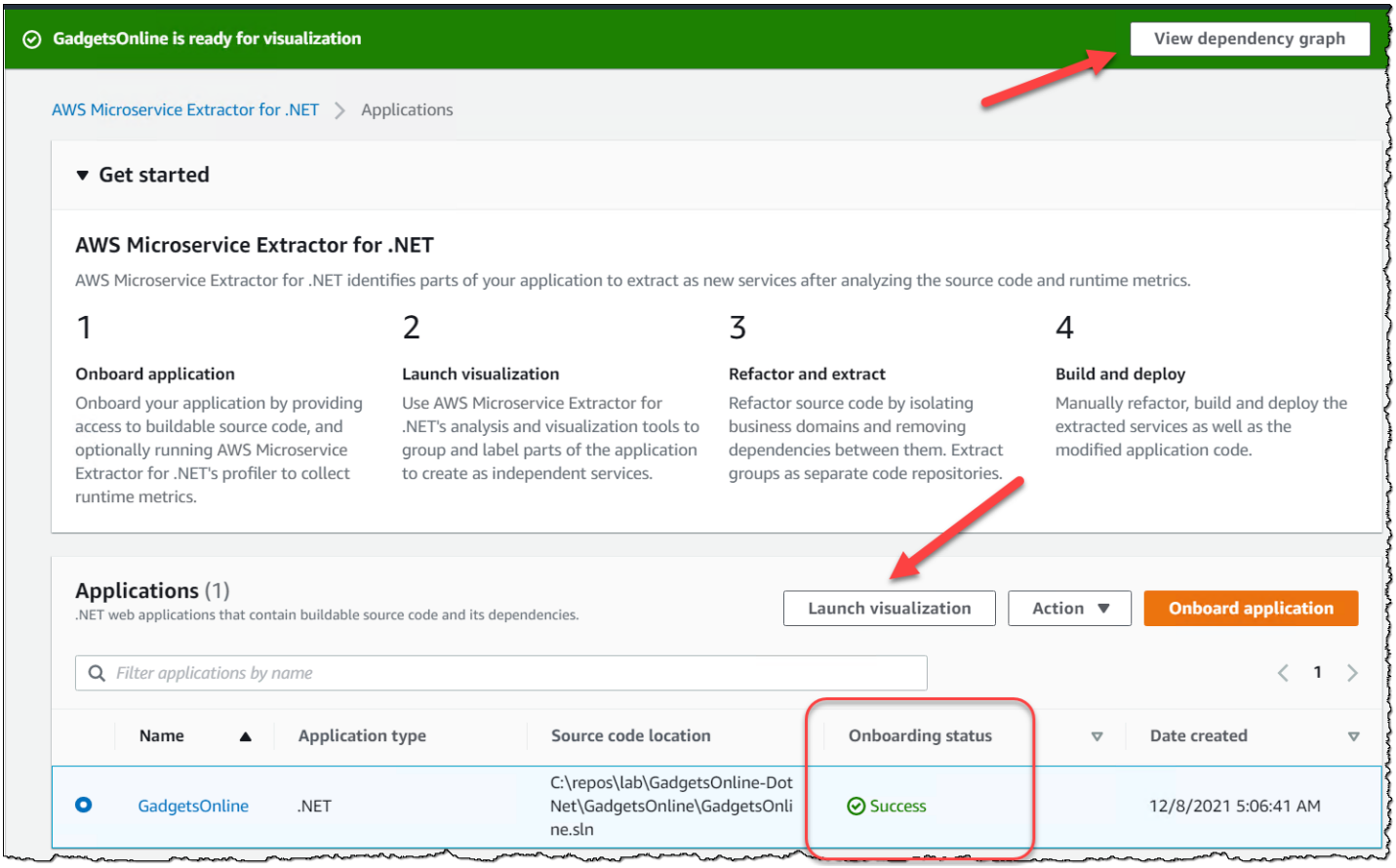
2. In the right pane, click on "**Onboard application**".

3. In the **Application details** page**;**
   - Provide N**ame** 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.

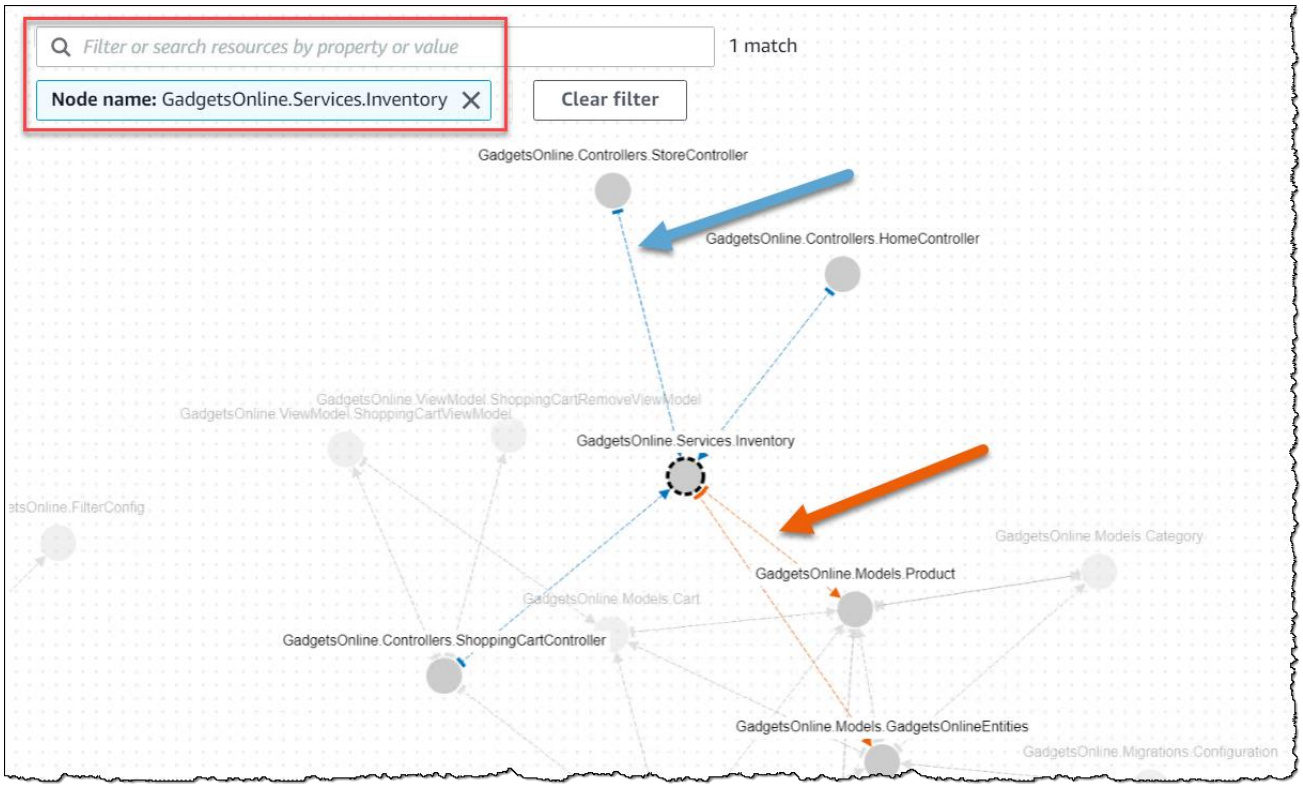
Launch Visualization

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.



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.

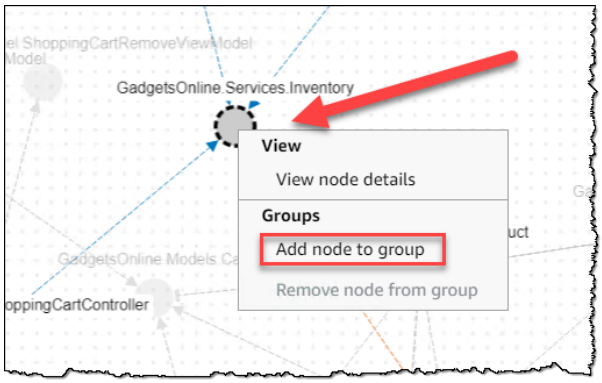
Before we go to next step and extract Inventory as a separate Microservice, it is worth reviewing various visualization options available with Microservice extractor. Review this blog to see **Namespace view** and **Island view** as alternate visualizations supported by Microservice extractor.

Create a group for extraction.

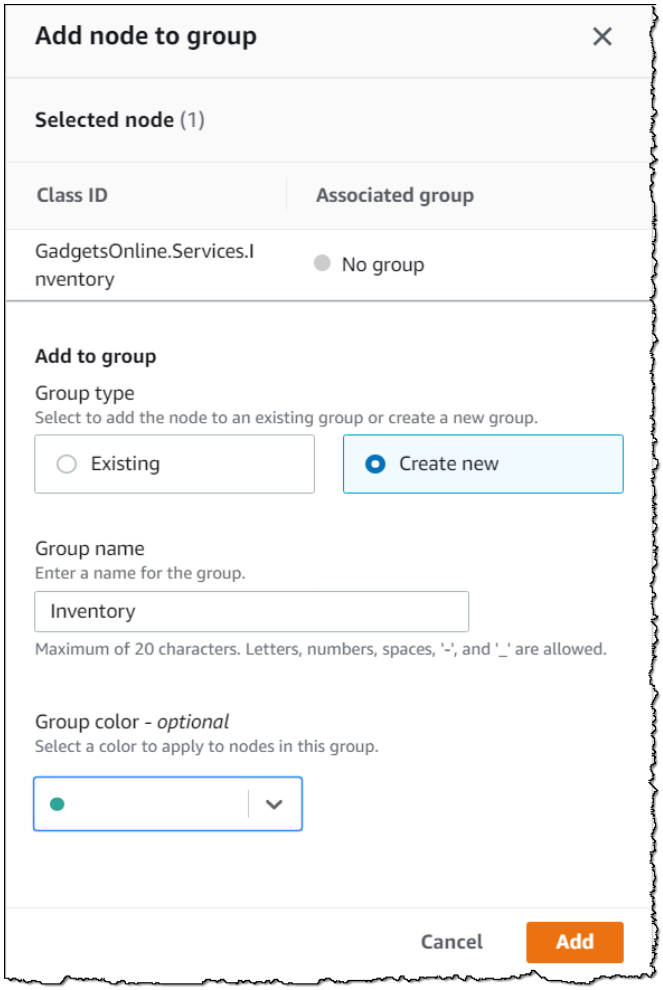
Once you visualize dependencies between various parts of your application, you can begin extracting the process. The extraction process requires organizing classes as a group which can be extracted as an independent service. This step might require refactoring your existing code before extraction.

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.**

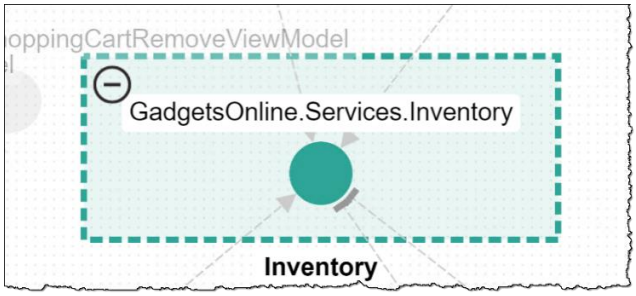


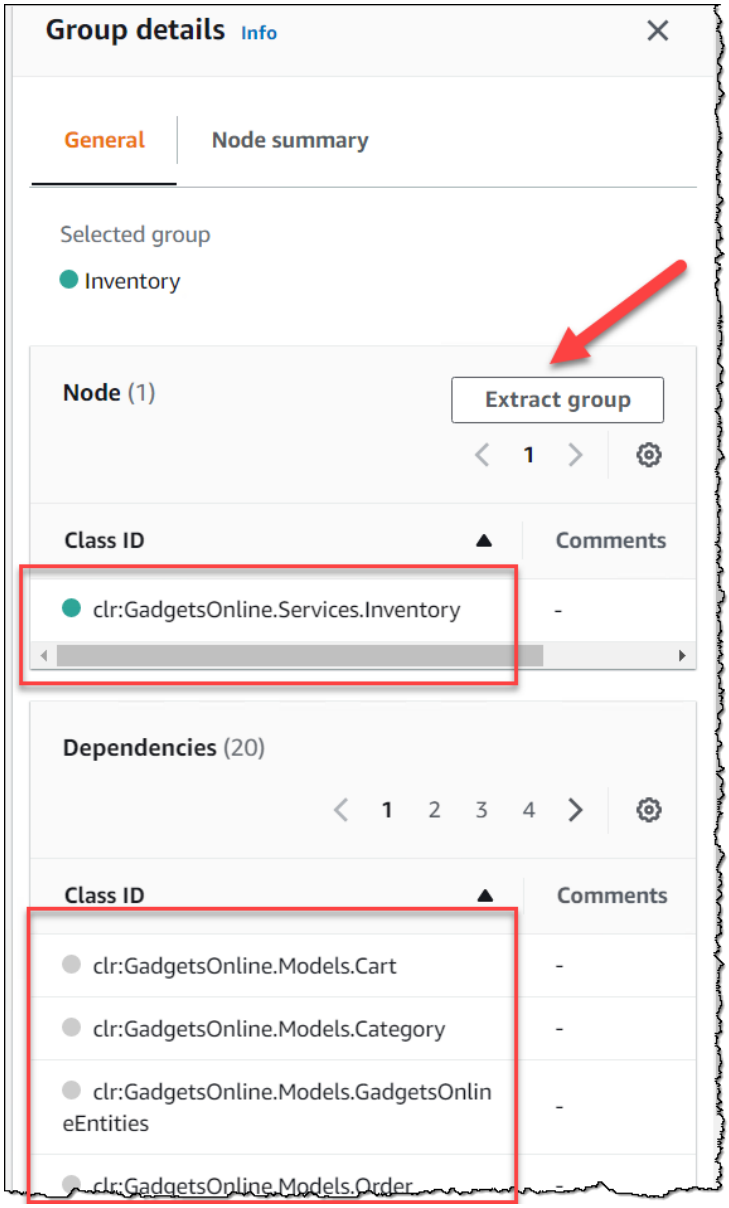
3. Select **Create new** to create a new group.
   Give **Group name** as **Inventory**.
   Choose a **Group Color** from the drop down and click **Add** button.



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.



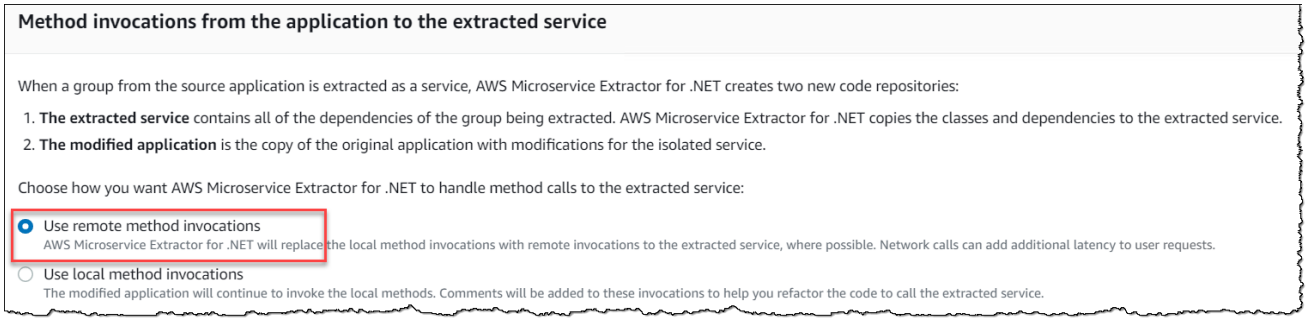


5. Click on **Extract group** button to begin extraction process.

Extract a microservice

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

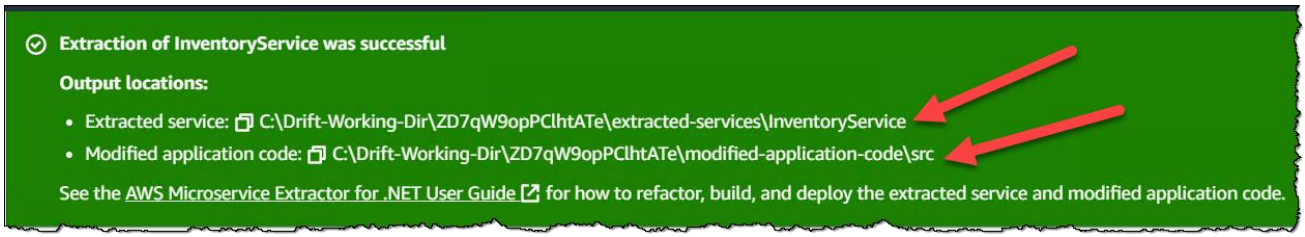
1. Provide service name as **InventoryService**
2. Review **Nodes** to be extracted. This should list **Inventory** class from the sample application.
   Review **Dependencies** list, these dependencies will be copied or referenced in the extracted independent service.
3. In the **Method invocation from 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.



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

Post Extraction

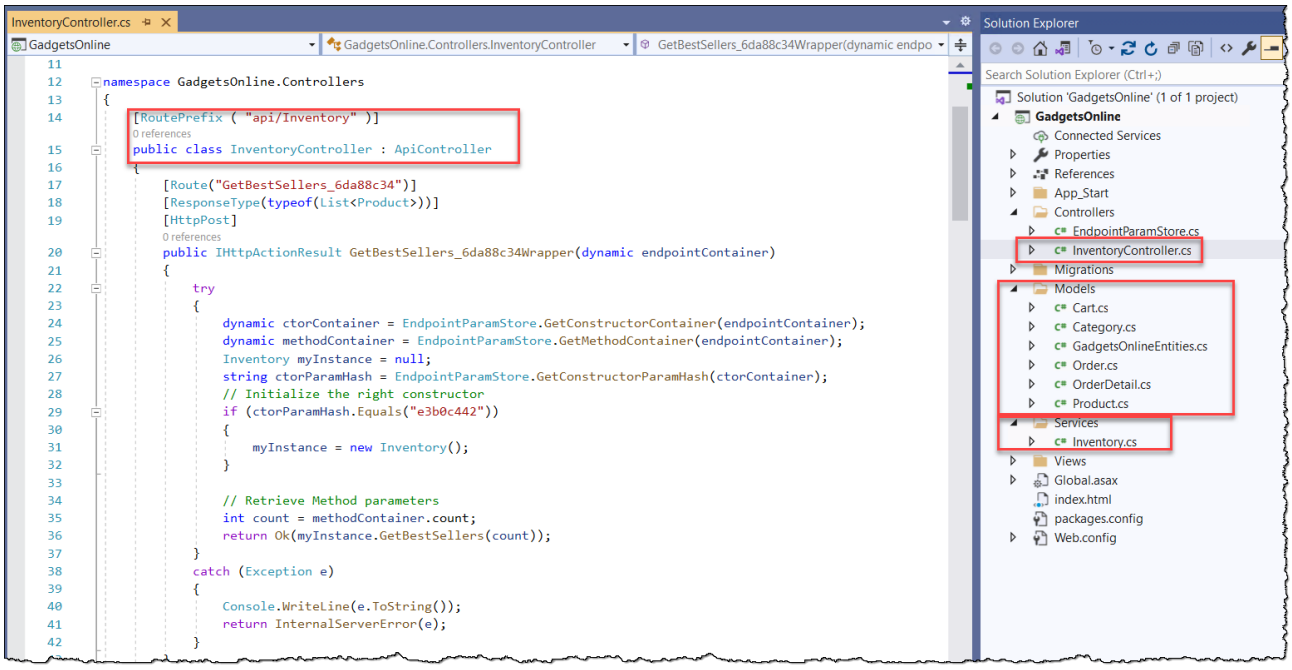
1. Once extraction is completed. You can copy the **Output locations** of the **Extracted service** and the **Modified application code.**



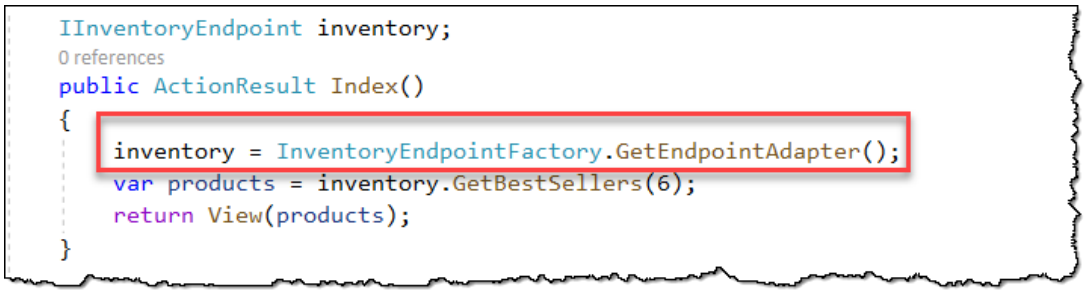
2. Double click the solution file in **Extracted service** folder to open the ASP.NET WebAPI project in visual studio and review the project structure.

   - 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.

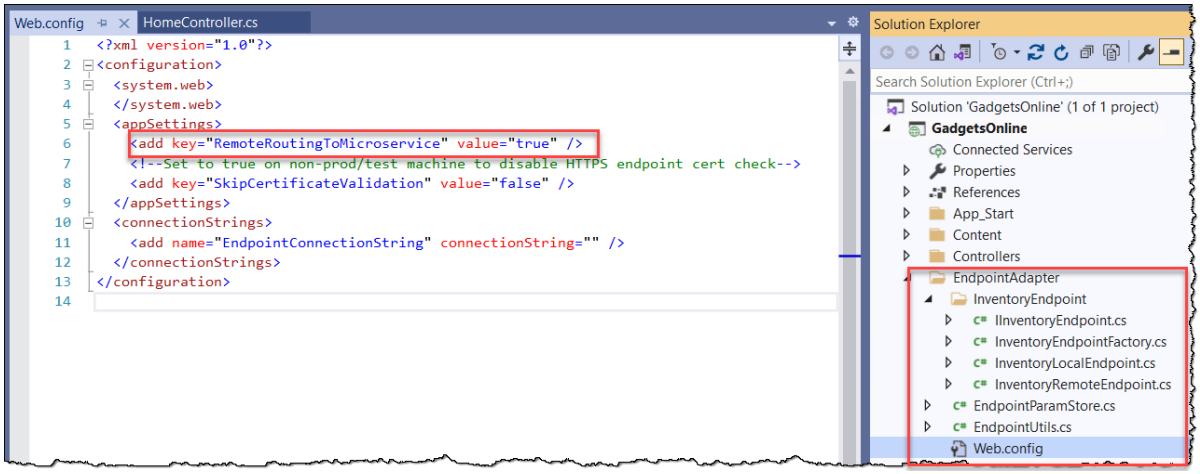
   Note that Microservice Extractor will make best efforts to making the newly created service compile able but it is not guaranteed. You may still have to fix references or update Nuget packages to compile the service.



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 as **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.