**.NetCore 2.2**

**DI – Config Settings – Logging**

Table of Contents

[Project 1: Configuration Settings 3](#_Toc21904944)

[Setting up Web Application 3](#_Toc21904945)

[Project in Solution Explorer 3](#_Toc21904946)

[Add a new Section to appsettings.json 4](#_Toc21904947)

[Create a Model 4](#_Toc21904948)

[Access it in the StartUp.cs 4](#_Toc21904949)

[Run the app and put a break point to take a look at the value 4](#_Toc21904950)

[How to get it via DI? 4](#_Toc21904951)

[Project 2: Dependency Injection – SimpleDB – Logging 5](#_Toc21904952)

[Project in solution explorer 5](#_Toc21904953)

[Creating a DB and Populating Tables 5](#_Toc21904954)

[View Model 7](#_Toc21904955)

[Setup without DI 7](#_Toc21904956)

[Mappers Class 7](#_Toc21904957)

[Repository 7](#_Toc21904958)

[BusinessLayer/ServiceClass 8](#_Toc21904959)

[HomeController / Index Action 9](#_Toc21904960)

[Index View 9](#_Toc21904961)

[Test with Production DB 9](#_Toc21904962)

[Complete Project View 10](#_Toc21904963)

[Setup with DI 10](#_Toc21904964)

[Moving the Connection String 10](#_Toc21904965)

[IConfigurationSettings & ConfigurationSettings – Interface and Class Setup 11](#_Toc21904966)

[ConfigurationSettings DI Setup 11](#_Toc21904967)

[Injecting ConfigurationSettings into Repository.cs 12](#_Toc21904968)

[Preparing Repository for DI 13](#_Toc21904969)

[Preparing CustomerService for DI 14](#_Toc21904970)

[Injecting CustomerService into HomeController 15](#_Toc21904971)

[Run the App again 15](#_Toc21904972)

[Project Overview with DI in place 16](#_Toc21904973)

[Using DI to Support Multiple Environment 16](#_Toc21904974)

[Current appsettings.json 16](#_Toc21904975)

[Current Content of appsettings.json 16](#_Toc21904976)

[Current content of appsettings.Development.json 17](#_Toc21904977)

[Building our own Configuration Model to work with different Environments 17](#_Toc21904978)

[Move the ConfigurationSettings from appsettings.json to appsettings.Development.json 17](#_Toc21904979)

[appsettings.json 17](#_Toc21904980)

[appsettings.Development.json 17](#_Toc21904981)

[Environment Settings with DI 18](#_Toc21904982)

[Properties\launchSettings.json 18](#_Toc21904983)

[Create Production Environment 18](#_Toc21904984)

[Run the App in Production Environment 19](#_Toc21904985)

[Logging with DI - NLog 20](#_Toc21904986)

[Install NLog.Web.AspNetCore 20](#_Toc21904987)

[NLog Setup 20](#_Toc21904988)

[Configure method updates 20](#_Toc21904989)

[Nlog.config 21](#_Toc21904990)

[NLog HomeController use 22](#_Toc21904991)

[Use of NLog in Repository Class 22](#_Toc21904992)

[NLog – different environments 23](#_Toc21904993)

[NLog Development 23](#_Toc21904994)

[NLog Production 24](#_Toc21904995)

[Launch Settings 24](#_Toc21904996)

[Log Email with DI 24](#_Toc21904997)

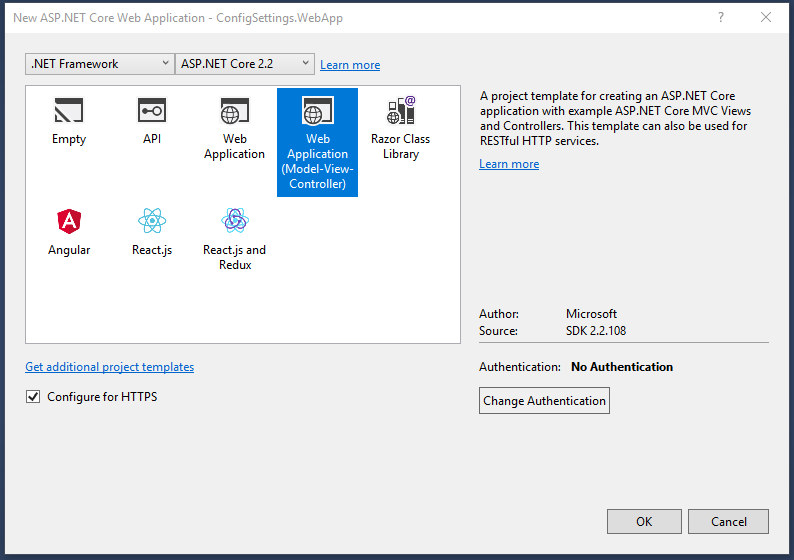
[Writing the NLog to SQL Server 25](#_Toc21904998)

[Table and StoredProcedure Create 25](#_Toc21904999)

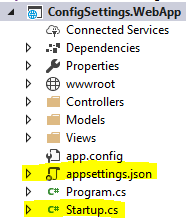
[New Targets and Rule 26](#_Toc21905000)

# Project 1: Configuration Settings

## Setting up Web Application



## Project in Solution Explorer

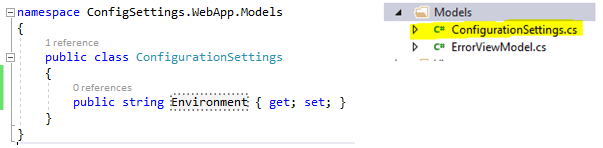


## Add a new Section to appsettings.json

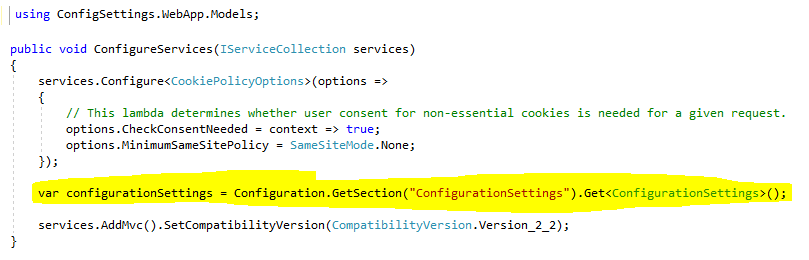


## Create a Model

Create a model that looks exactly like a configuration setting we added above



## Access it in the StartUp.cs



## Run the app and put a break point to take a look at the value



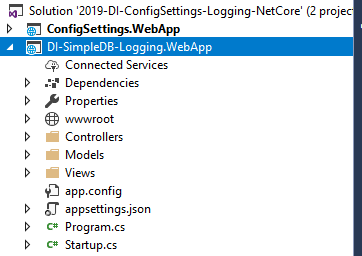
## How to get it via DI?

Go to next project to learn about DI

# Project 2: Dependency Injection – SimpleDB – Logging

Create a WebApp just like in [project 1](#_Setting_up_Web) and name it DI-SimpleDB-Logging.WebApp and set it as a startup project.

## Project in solution explorer



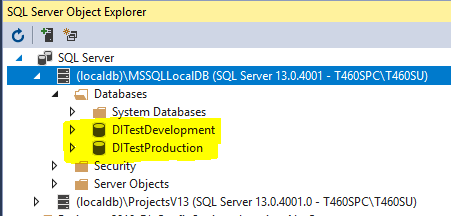
## Creating a DB and Populating Tables

Since we will be pulling data from the db in this example, please perform the following steps

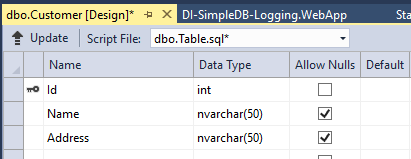
1. Install SQL Server Express / SQL Server Management Studio Express
2. Using SQL Server Object Explorer create two databases for development and production

Since I am using LocalDB, the DBs will get created in the following location

C:\Users\USER\AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances\mssqllocaldb



Then create a Customer table in both the DB and populate with different values



CREATE TABLE [dbo].[Customer]

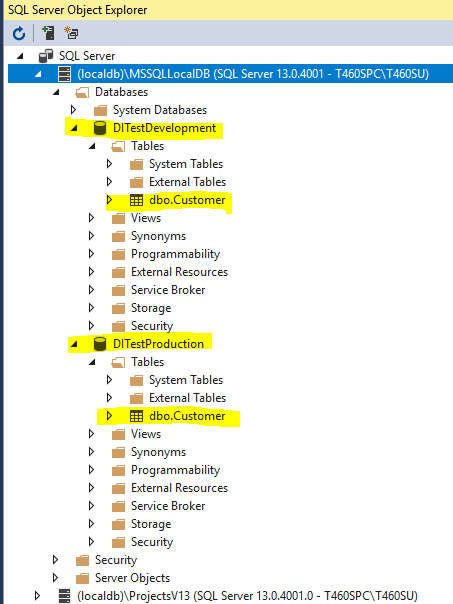
(

[Id] INT NOT NULL IDENTITY PRIMARY KEY,

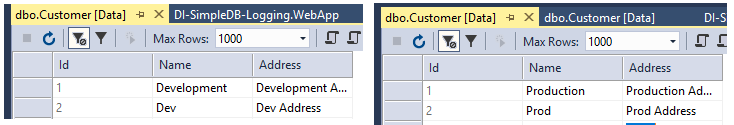
[Name] NVARCHAR(50) NULL,

[Address] NVARCHAR(50) NULL

)

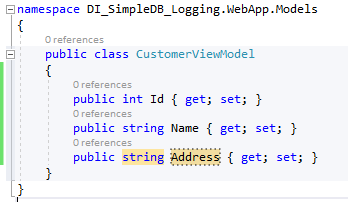


Add the data to both the tables



## View Model

Create a view model in the Model folder to retrieve the results



## Setup without DI

### Mappers Class

Create a mapper class in the “Core” folder to map Customer table records to Customer class objects



### Repository

Create a generic Repository class in the “Core” folder. It will use the SqlCommand to get the data. A mapping function will be passed to this generic method to retrieve the data. In a real world application, we’ll have a business layer which will call the repository. For the [connection info](#_Creating_a_DB), check above.



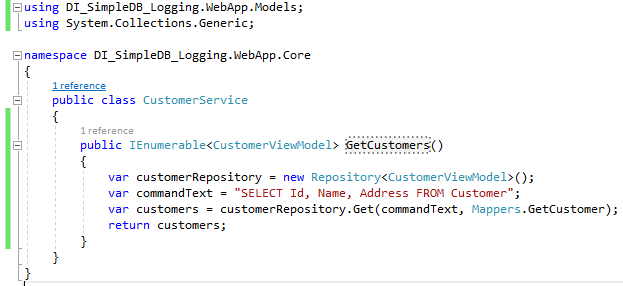
Connection string used by the image above

//DevDB=DITestDevelopment; ProdDB=DITestProduction

var connectionString = "Server=(localdb)\\mssqllocaldb;Database=DITestDevelopment;Trusted\_Connection=True;MultipleActiveResultSets=true";

### BusinessLayer/ServiceClass

We’ll create a service class which will in turn call the Repository Get method

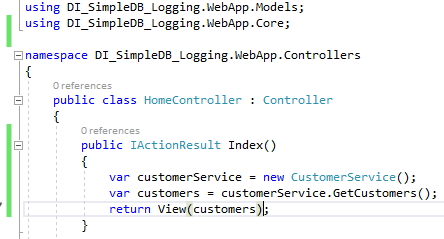


The SQL statement used by the above function

var commandText = "SELECT Id, Name, Address FROM Customer";

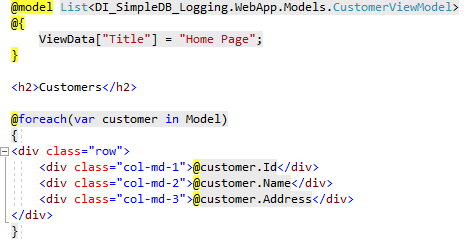
### HomeController / Index Action

We’ll use CustomerService to get the customers in the Index action of the HomeController.

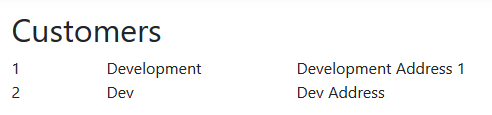


### Index View

The view we bind to is Views\Home\Index.cshtml. Open this file and replace the contents of it with the following.



And the view result looks like



### Test with Production DB

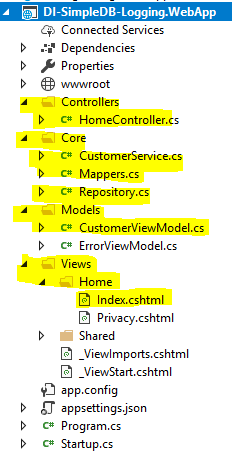
Go to Repository.cs and change the connection string to following and test again

var connectionString = "Server=(localdb)\\mssqllocaldb;Database=DITestProduction;Trusted\_Connection=True;MultipleActiveResultSets=true";

This time the records should display from Production DB



### Complete Project View

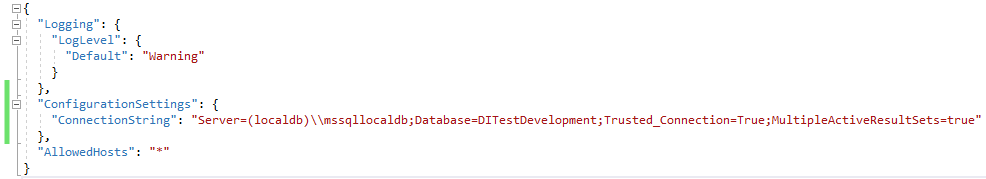


## Setup with DI

Now we’ll convert the project to use Dependency Injection.

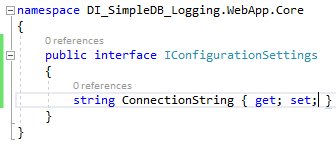
### Moving the Connection String

We created the connection string in Repository.cs. We need to move that to the appsettings.json. We’ll also create a different json file for development and production. Change the appsettings.json to following by moving the connection string from [Repository.cs](#_Repository) to it. Here we’ll create a ConfigurationSettings section and then will move the connection string to it.

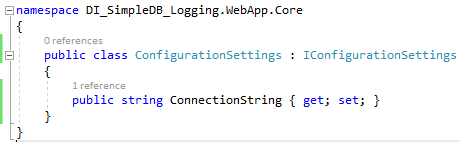


### IConfigurationSettings & ConfigurationSettings – Interface and Class Setup

In order to use DI with configuration settings, first create a IConfigurationSettings interface with one property inside the “Core” folder.



And then create a ConfigurationSettings class inside the “Core” folder and implement interface IConfigurationSettings.



### ConfigurationSettings DI Setup

Open **Startup.cs** in the root and then go to the ConfigureServices method.

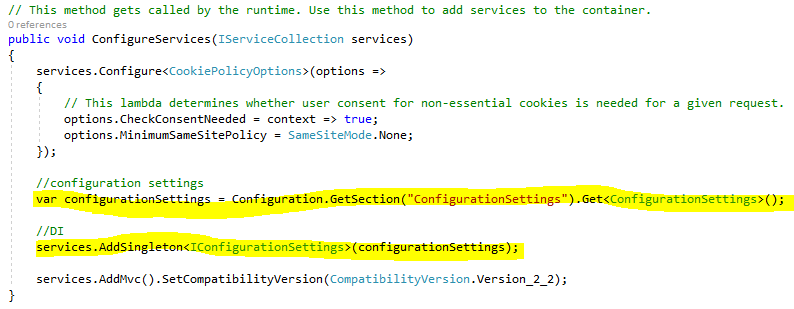
The Configuration class is built into it and is used to map the json to ConfigurationSettings class. DI starts here, this is where we’ll bind the interface to an implementation. Then throughout the application whenever we inject the instance of an interface the implementation will get bound to it.

We are using the AddSingleton which will create the single instance of the class to inject. In other cases we’ll use AddTransient which will produce the new instance of the class every time we inject it into the class.

First put the using statement



And then add the two lines of code in ConfigureServices.



### Injecting ConfigurationSettings into Repository.cs

Go to Repository.cs in the “Core” folder and make the following changes

1. Create a property ConfigurationSettings
2. Add a constructor with a parm of type IConfigurationSettings and then assign this to the property.
3. Use the property to get the connection string.

Remember that we need to remove following

//DevDB=DITestDevelopment; ProdDB=DITestProduction

var connectionString = "Server=(localdb)\\mssqllocaldb;Database=DITestDevelopment;Trusted\_Connection=True;MultipleActiveResultSets=true";

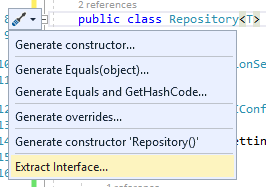
The updated Repository.cs class should look like



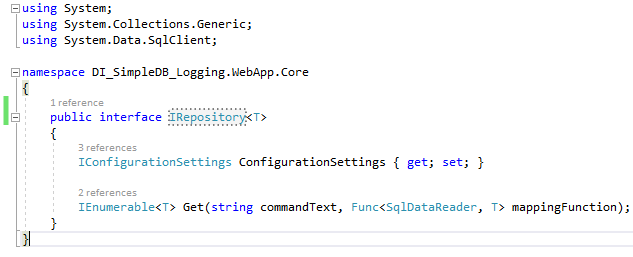
### Preparing Repository for DI

We are not done yet, we need to inject Repository into CustomerService

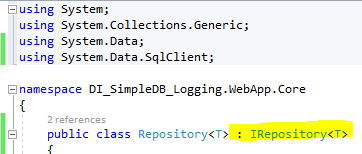
The easy way to create an interface for the Repository is to place the cursor at the end of Repository and then on keyboard press CTRL+. And then select “Extract Interface”.



The interface will look like



And the class will now look like

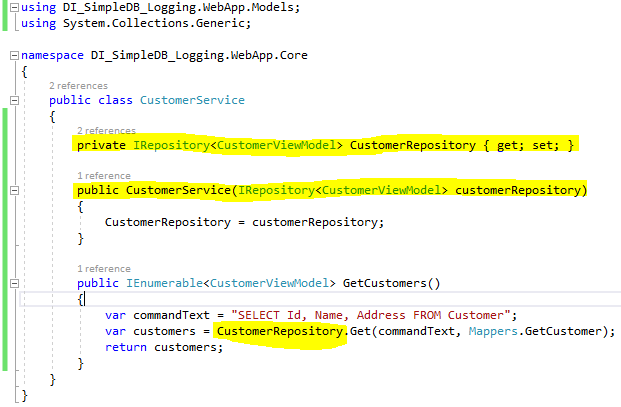


Next go to Startup.cs class and then add the DI for Repository in ConfigureServices method



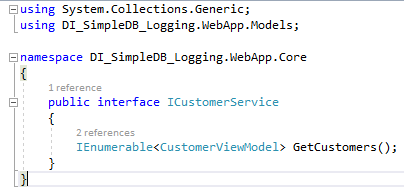


Then go to CustomerService and inject Repository into it.

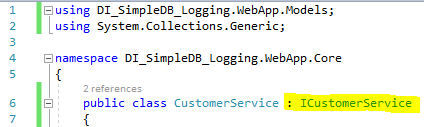


### Preparing CustomerService for DI

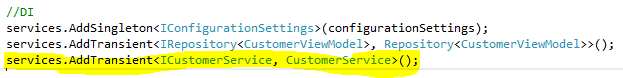
First create an interface for CustomerService by opening the CusomterService and placing the cursor at the end of CustomerService class name and then pressing CTRL+. and then selecting “Extract Interface”. The interface will look like



And the CustomerService class will look like following with the interface implemented

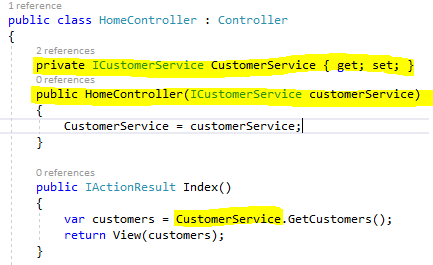


Then go to Startup.cs and add the DI for CustomerService to ConfigureServices method.



### Injecting CustomerService into HomeController

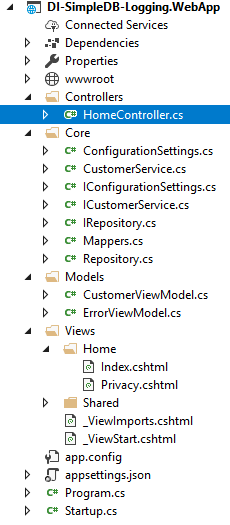
We need to inject the CustomerService into the HomeController which will now look like following with CustomerService injected.



### Run the App again

With all these changes in place, run the app with “DITestDevelopment” connection in the appsettings.json and then with “DITestProduction”. The app should work as expected.

### Project Overview with DI in place



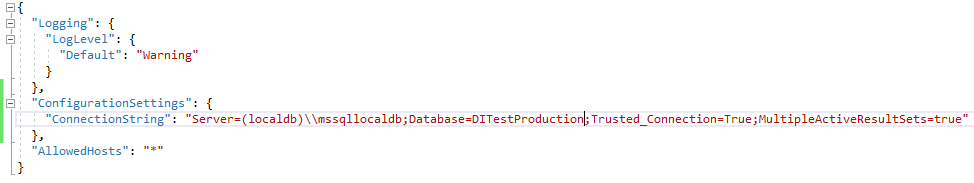
## Using DI to Support Multiple Environment

### Current appsettings.json

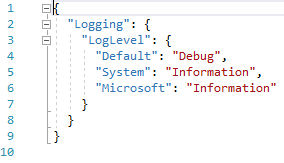
Expand the appsettings.json node and it will look like following



#### Current Content of appsettings.json

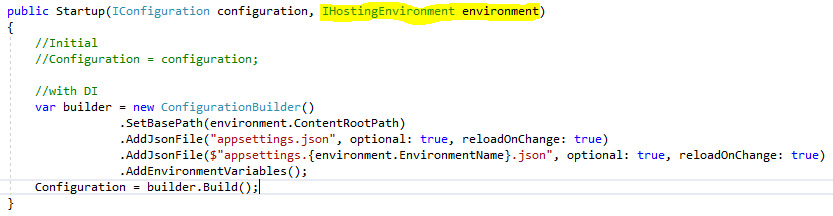


Current content of appsettings.Development.json



### Building our own Configuration Model to work with different Environments

Go to Startup.cs and then replace the contents of the constructor with following



It will first try to load the settings from appsettings.json file.

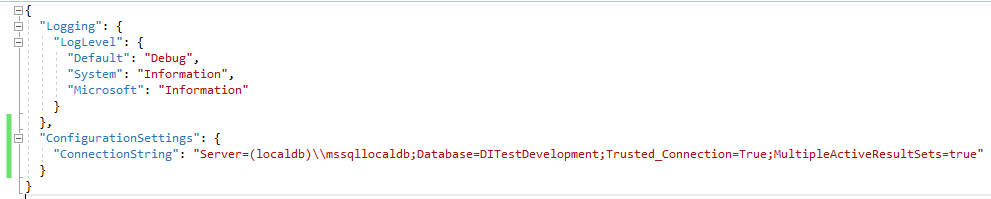
Then it will try to load the settings from the environment.

### Move the ConfigurationSettings from appsettings.json to appsettings.Development.json

#### appsettings.json



#### appsettings.Development.json



And run the app, it should work as usual.

### Environment Settings with DI

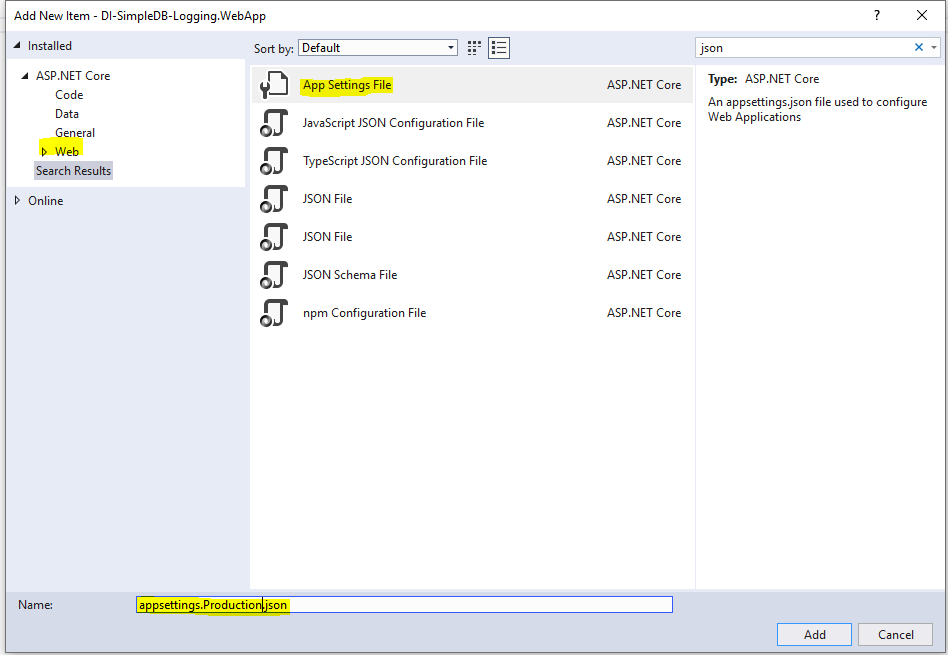
How does the application know that it is running in the development environment? The answer is very simple… It looks at the Properties\launchSettings.json file.

#### Properties\launchSettings.json



#### Create Production Environment

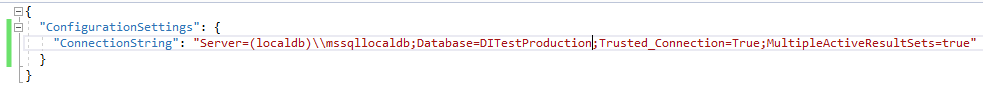
Add a new appsettings json file and name it appsettings.Production.json



The appsettings will look like



Open the new production appsettings json file add the production connection string to it.



### Run the App in Production Environment

Open Properties\launchSettings.json and replace “Development” with “Production”



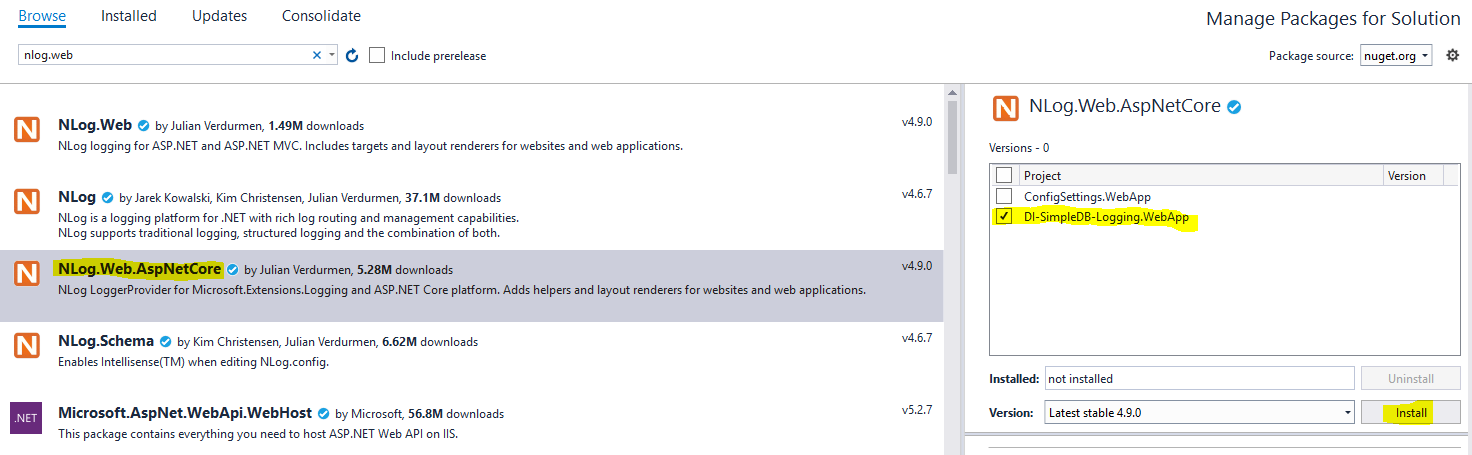
Run the app and you should now see your production data.

## Logging with DI - NLog

.Net Core has a built in logging framekwork but to take advantage of it, we’ll use NLog to write to text file, email and write to the DB.

### Install NLog.Web.AspNetCore

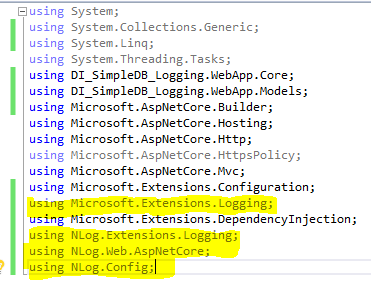
Install the nuget package NLog.Web.AspNetCore



### NLog Setup

We’ll set it up through the startup.cs, setting it here will help us inject into controllers, services and repositories etc.

Few references that we’ll need to add

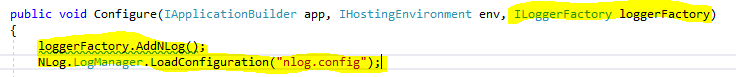


#### Configure method updates

We’ll need to make changes to the **Configure** method. .Net Core has built in factory class to support logging. What the factory class does is that it will produce a logger for us.

#### Nlog.config

We need to give the factory the ability to produce an NLog. First we’ll tell the factory class to add NLog and then to use the **nlog.config**.



But we cannot use the config till we created it. We’ll create it in the root folder of our project. Add the following config to the nlog.config.

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<target name="loggingFile" xsi:type="File" fileName="c:\\temp\dev.log"

layout="${date}|${level:uppercase=true}|${message} ${exception}|${logger}|${all-event-properties}">

</target>

</targets>

<rules>

<logger name="\*" minlevel="Warn" writeTo="loggingFile" />

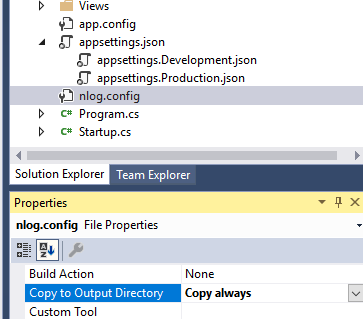
</rules>

</nlog>

Since we are trying to set it up first, the barebone version has two elements, targets and rules.

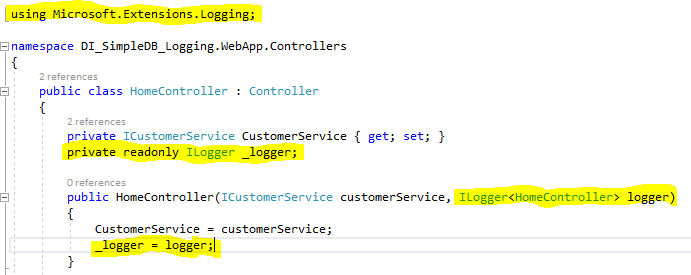
* The target is where the logging data will go. This could be the database or the email or the file. For now, we’ll just write to the text file.
* The rules is the message levels we are interested in. there are a lot of message levels like
  + Info
  + Debug
  + Warn – we’ll only log warning messages
  + Error

Open the properties of this new nlog.config file and change “Copy to Output Directory” to “Copy always”.

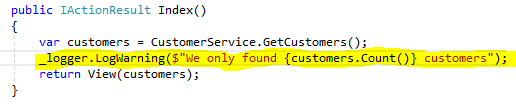


### NLog HomeController use

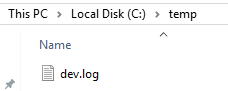
Open HomeController, our NLog is not available for injection into home controller. Inject it into HomeController constructor.



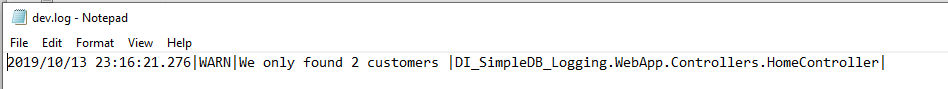
Then add a statement to log the warning inside the index action.



Run the app and look for the log file in **c:\\temp** folder.

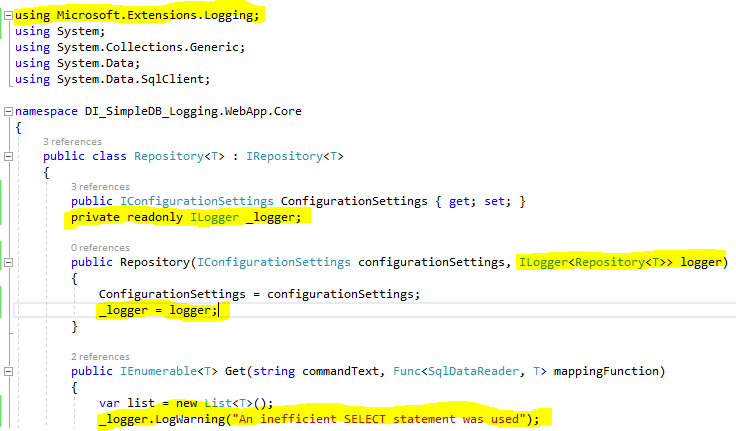


Open the log to view its content.

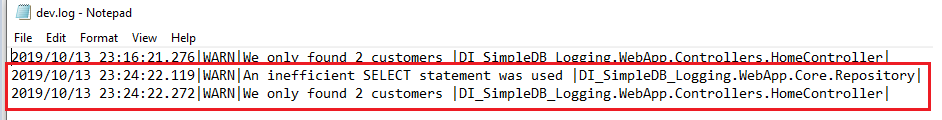


### Use of NLog in Repository Class

Now inject the logger in the repository class



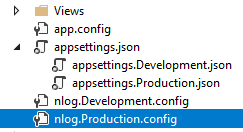
Run the app and then look at the log file created



### NLog – different environments

We can switch between the different environments with ease.

Change the nlog.config to nlog.Development.config. and also create a new one with name nlog.Production.config. Introduce some difference between the two.



#### NLog Development

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<target name="loggingFile" xsi:type="File" fileName="c:\\temp\development.log"

layout="${date}|${level:uppercase=true}|${message} ${exception}|${logger}|${all-event-properties}">

</target>

</targets>

<rules>

<logger name="\*" minlevel="Warn" writeTo="loggingFile" />

</rules>

</nlog>

#### NLog Production

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<target name="loggingFile" xsi:type="File" fileName="c:\\temp\production.log"

layout="${date}|${level:uppercase=true}|${message} ${exception}|${logger}|${all-event-properties}">

</target>

</targets>

<rules>

<logger name="\*" minlevel="Warn" writeTo="loggingFile" />

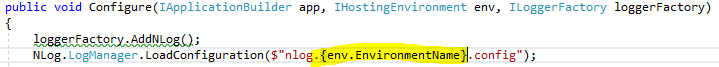
</rules>

</nlog>

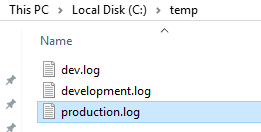
#### Launch Settings

Change the environment name from “Development” to “Production”

And then back in the startup.cs\Config method pick the file per the environment



And run the app again and the look at the **c:\\temp** folder.



## Log Email with DI

Open up the nLog.ENVIRONMENT.config and setup the email

I have created a new email which will be used to send the email. For testing purposes, I created gmail account with name “nLogTest1013”.

Go back into nlog.Production.log file and add a new target and rule for sending the emails.



Here the production will send the email where as the development will write to the text file. Run the app in production mode by changing the environment to Production in Properties\launchSettings.json file.

## Writing the NLog to SQL Server

For more info:

<https://github.com/nlog/NLog/wiki/Database-target>

### Table and StoredProcedure Create

In this example will be using the stored procedure to write the log.

CREATE TABLE [dbo].[NLog] (

[ID] [int] IDENTITY(1,1) NOT NULL,

[MachineName] [nvarchar](200) NULL,

[Logged] [datetime] NOT NULL,

[Level] [varchar](5) NOT NULL,

[Message] [nvarchar](max) NOT NULL,

[Logger] [nvarchar](300) NULL,

[Properties] [nvarchar](max) NULL,

[Callsite] [nvarchar](300) NULL,

[Exception] [nvarchar](max) NULL,

CONSTRAINT [PK\_dbo.Log] PRIMARY KEY CLUSTERED ([ID] ASC)

WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY];

GO

CREATE PROCEDURE [dbo].[NLog\_AddEntry\_p] (

@machineName nvarchar(200),

@logged datetime,

@level varchar(5),

@message nvarchar(max),

@logger nvarchar(300),

@properties nvarchar(max),

@callsite nvarchar(300),

@exception nvarchar(max)

) AS

BEGIN

INSERT INTO [dbo].[NLog] (

[MachineName],

[Logged],

[Level],

[Message],

[Logger],

[Properties],

[Callsite],

[Exception]

) VALUES (

@machineName,

@logged,

@level,

@message,

@logger,

@properties,

@callsite,

@exception

);

END

### New Targets and Rule

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<target name="gmail" xsi:type="Mail"

smptpServer="smtp.gmail.com"

smtpPort="587"

smtpAuthentication="Basic"

smtpUserName="nLogTest1013@gmail.com"

smtpPassword="..."

enableSsl="true"

from="nLogTest1013@gmail.com"

to="nLogTest1013@gmail.com"

subject="Production Warning Message"

/>

<target name="db"

xsi:type="Database"

connectionString="Server=(localdb)\\mssqllocaldb;Database=DITestProduction;Trusted\_Connection=True;MultipleActiveResultSets=true"

commandType="StoredProcedure"

commandText="[dbo].[NLog\_AddEntry\_p]"

>

<parameter name="@machineName" layout="${machinename}" />

<parameter name="@logged" layout="${date}" />

<parameter name="@level" layout="${level}" />

<parameter name="@message" layout="${message}" />

<parameter name="@logger" layout="${logger}" />

<parameter name="@properties" layout="${all-event-properties:separator=|}" />

<parameter name="@callsite" layout="${callsite}" />

<parameter name="@exception" layout="${exception:tostring}" />

</target>

</targets>

<rules>

<logger name="\*" minlevel="Warn" writeTo="gmail" />

<logger name="\*" minlevel="Warn" writeTo="db" />

</rules>

</nlog>