**Assignment**

Develop a .NET Core API using docker Container. Database: Make a student table. With StudentID, Roll No, First Name and Last Name. Add some dummy records.

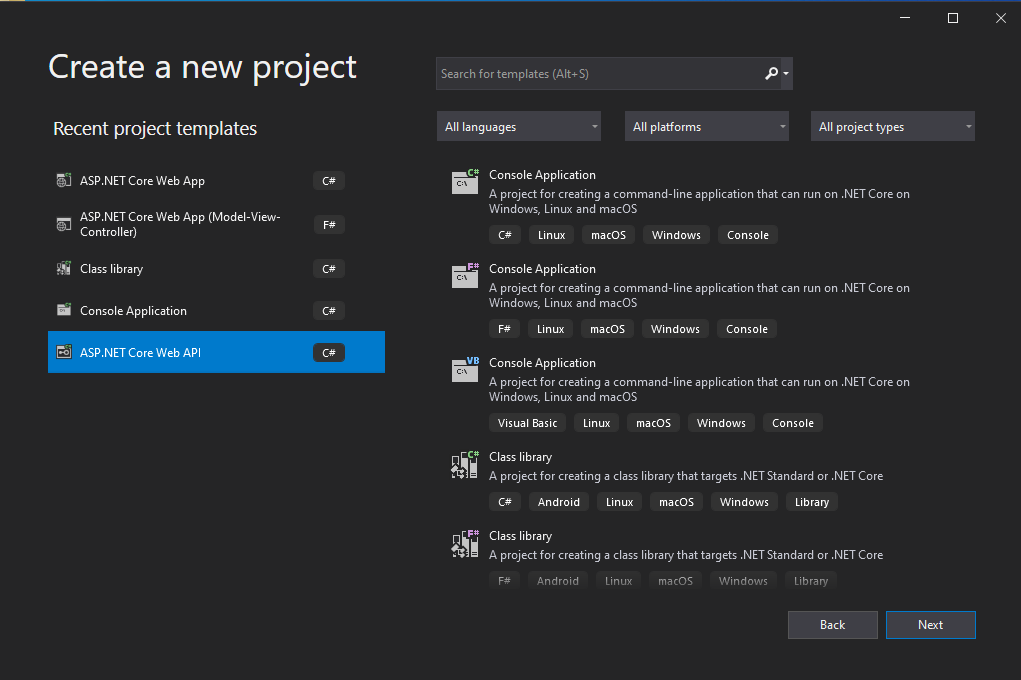
**Methods in API:**

1. *Get List of students in JSON Format.*
2. *Get Specific student by StudentID.*
3. *POST Method for inserting a new student record.*
4. *Update (PUT) a student record.*
5. *Delete a record from student Table.*

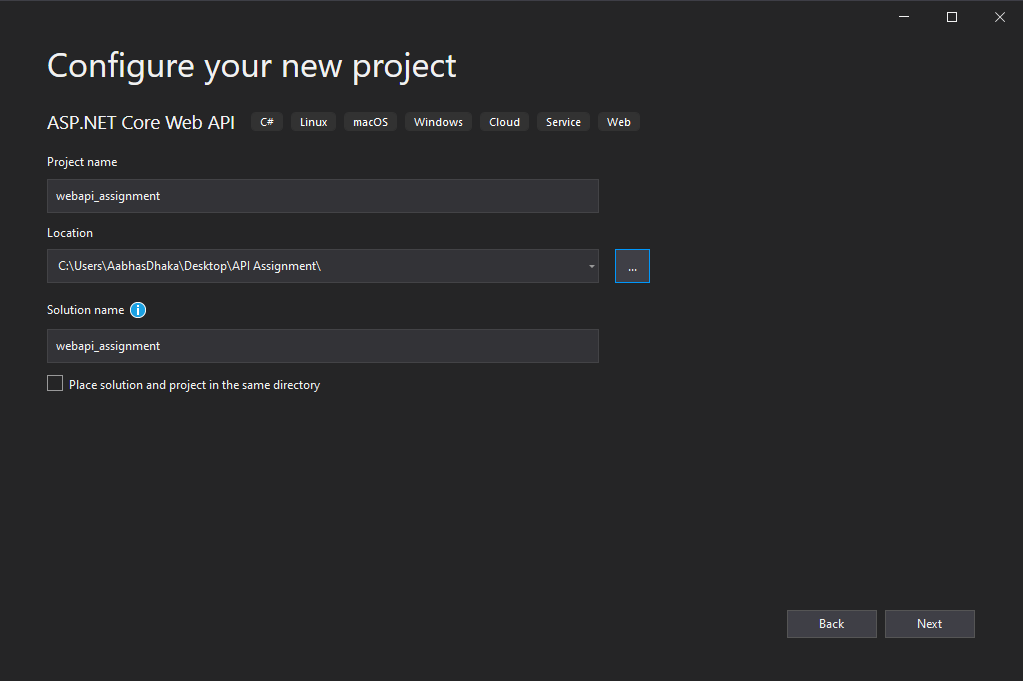
The minimum method which are expected in API is one get, and one post and maximum is all the Methods. For now, you can use either ADO.NET or Entity Framework for Data Access which ever helps you to achieve the object quicker. Demo and code walkthrough to be provided at the end of the assignment.

**Methodology Followed to achieve the above objectives.**

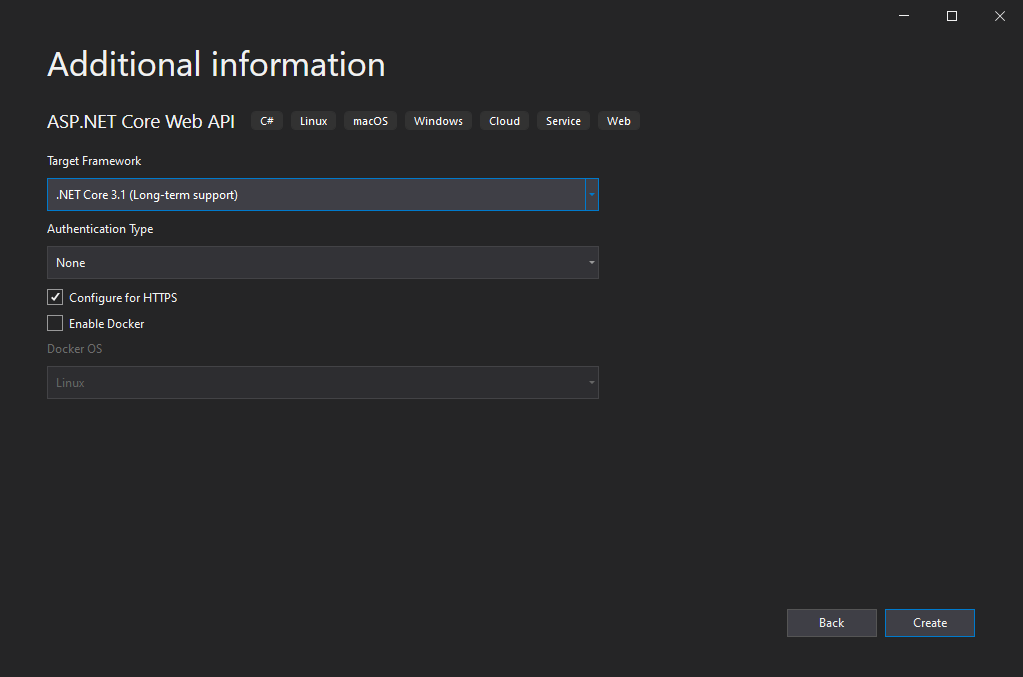
1. Creating a sample Asp.net core web API using ASP.NET Core Web API Template from Visual Studio.



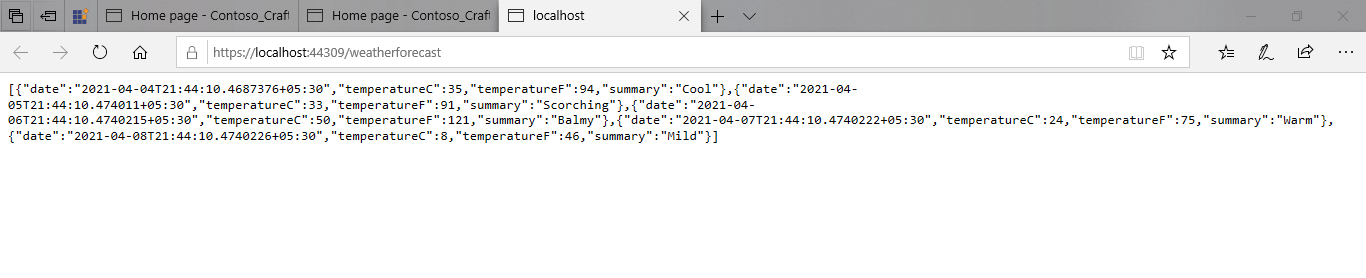
1. Creating the project with the name of *webapi\_assignment*.



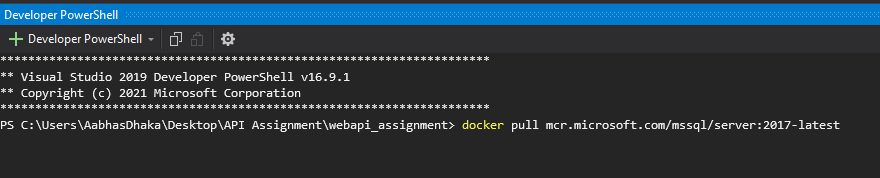
1. Using .NET Core 3.1 that is configured with https.



1. Checking if the Sample Project works by Running the project without Debugging mode and navigating to the specified URL in the Browser.



1. Knowing that the template works, a terminal is opened inside visual studio and mssqlserver:2017 image is pulled using docker.

****

1. After pulling the image, a container is made for the mssqlserver:2017 image using the *docker run* command.

**Command:**

*docker run -e "ACCEPT\_EULA=Y" -e "SA\_PASSWORD=Sample123$" -p 1433:1433 --name sqlserver2017withoutmount -d mcr.microsoft.com/mssql/server:2017-latest*

* -e 'ACCEPT\_EULA

Sets the ACCEPT\_EULA variable to ‘Y’ to confirm the acceptance of the End-User Licensing Agreement.

* -e 'SA\_PASSWORD

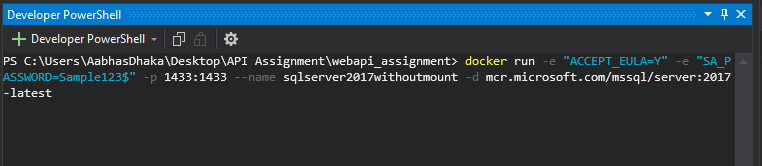
Password for login into SQL instance using *sa* username.

* -p 1433:1433

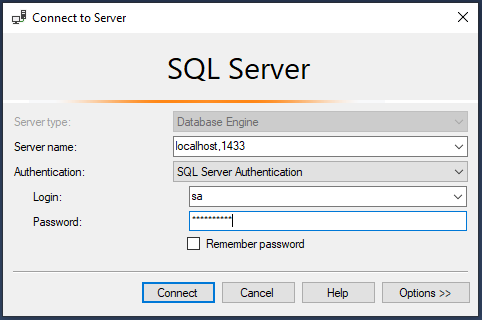
Map a TCP port on the host environment (first value) with a TCP port in the container (second value). In this example, SQL Server is listening on TCP 1433 in the container, and this is exposed to the port, 1433, on the host.

* --name

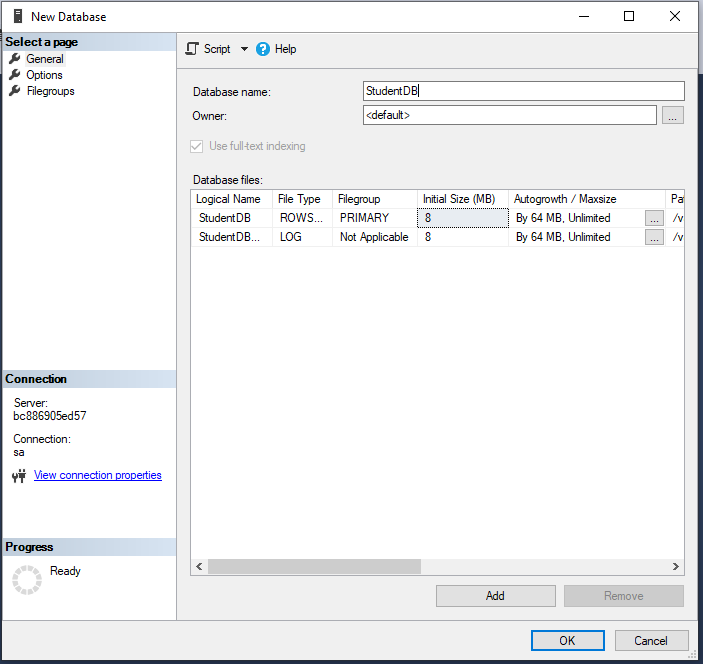
Specifies a name for the container rather than a randomly generated one.



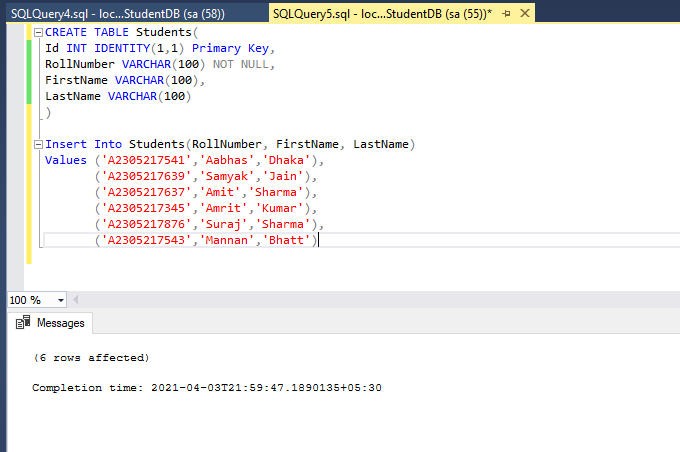
1. After the SQL Server Container is deployed, using Microsoft SQL Server Management Studio, a connection is made to the SQL Server using the credentials.



1. Now a Database is created by the name of *StudentDB* inside the Server*.*



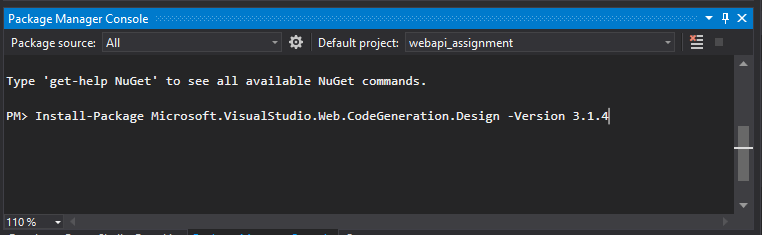
1. After the Database has been created, we use a query to a table named *Students* with the respective fields inside it. Also, some dummy records are inserted inside the table.



1. After completing the database part, we need to install some dependencies inside the project. These dependencies are installed using the Nu-Get Package Manager Console.

* The first dependency to be installed is***Microsoft.Visualization.Web.CodeGeneration.Design.***

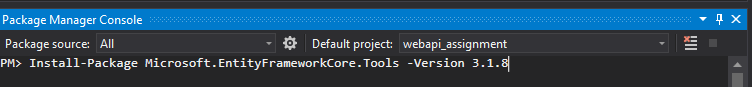
This Package helps in generating controllers and views.



* The Second Dependency to be installed is

***Microsoft.EntityFrameworkCore.Tools***.

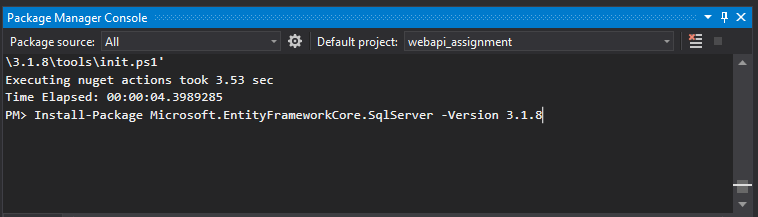
This package helps in creating database context and model classes from the database.



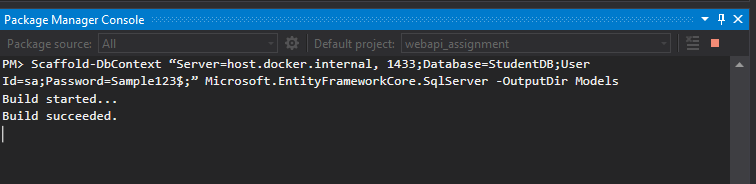
* The Third Dependency to be installed is

***Microsoft.EntityFrameworkCore.SqlServer***.

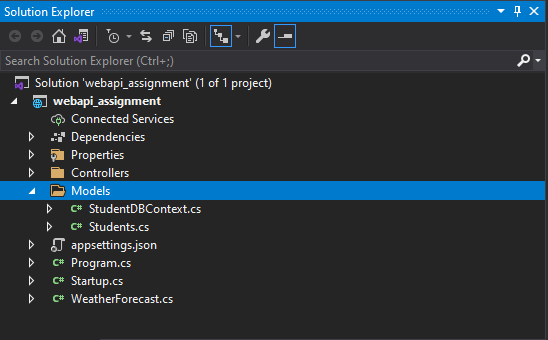
This package allows entity framework to work with SQL server.



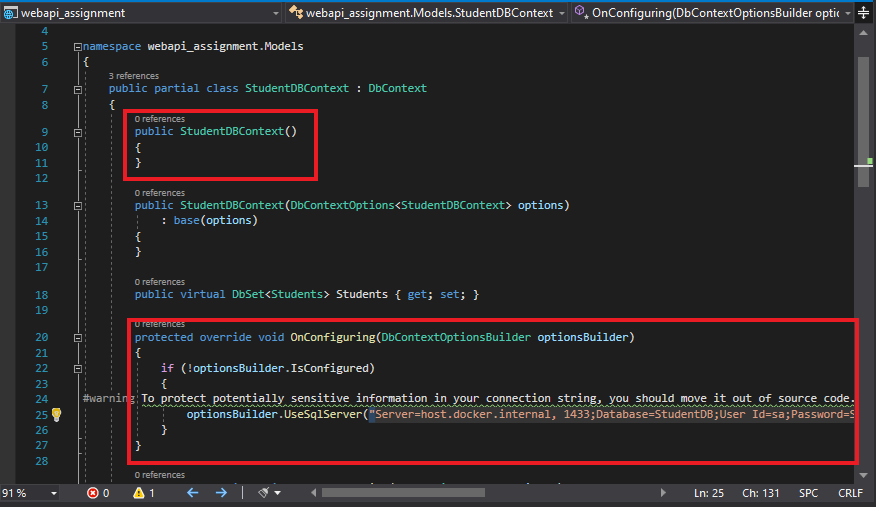
1. Now we need to scaffold the database context and entity POCO classes from the tables present inside the SQL Server. This could be done by using the above package installed and specifying the Connection String in a command inside the Nu-Get Package Manager Console.



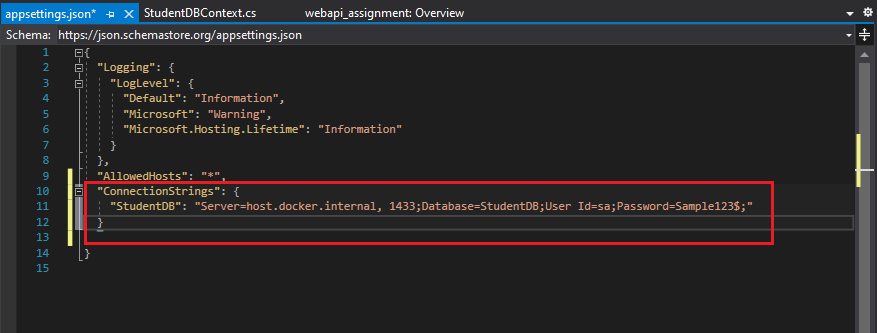
1. After issuing this command to the Terminal, a *StudentDBContext.cs* and a *Students.cs* file is created inside the *Models* folder in the solution.



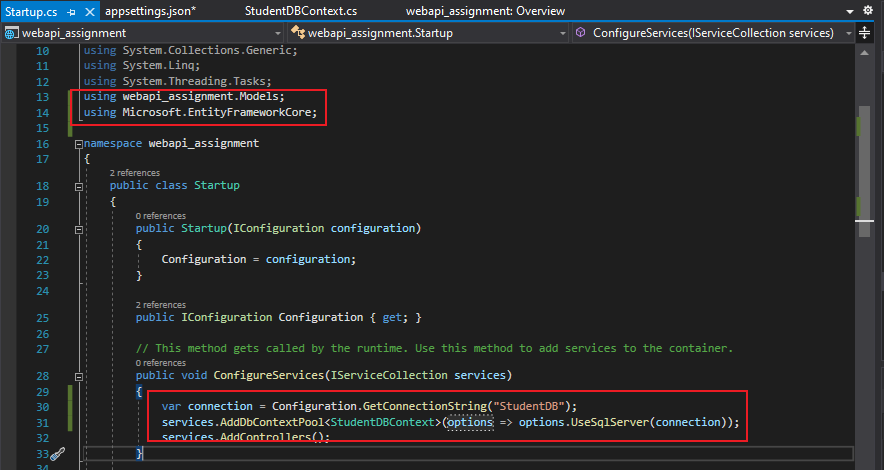
1. On viewing the *StudentsDBContext.cs* file, it is found out that the database credentials are hard coded in the *OnConfiguring* method. It is not a good practice, so we remove this method along with the parameterless constructor.

****

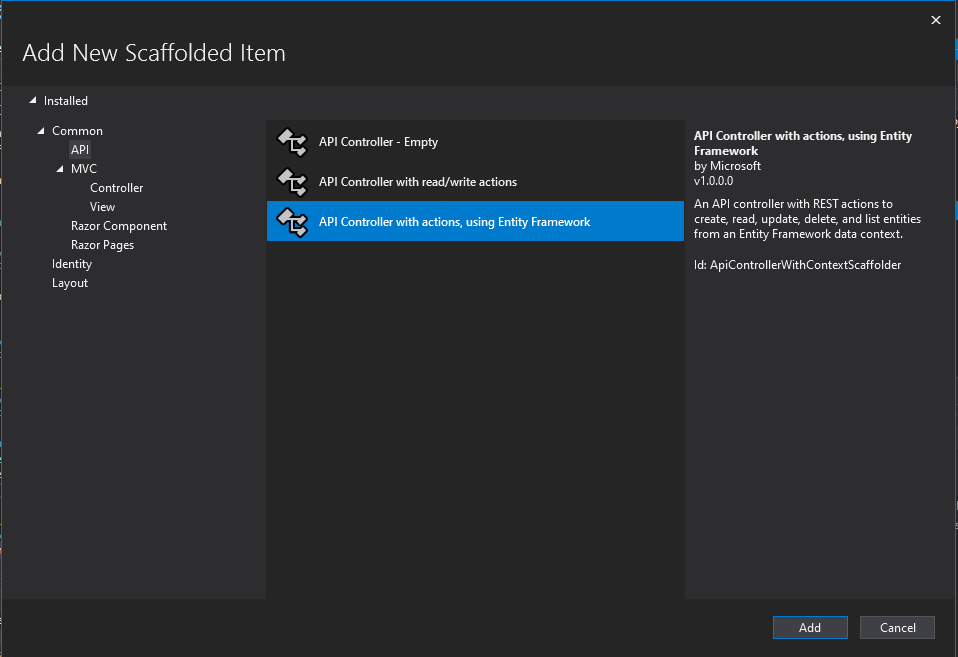
1. Instead, we add that connection string to the *appsettings.json* file.

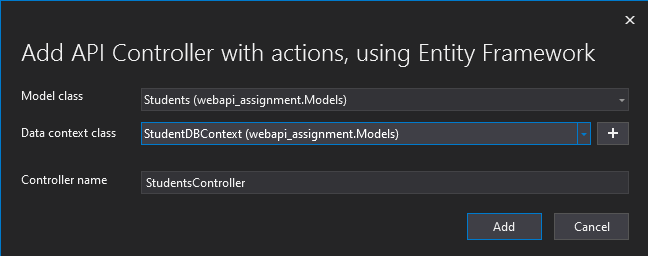


1. Now we need to add a service to our startup file so that it could be used anywhere in our application using dependency injection. We need to mention the essential packages that are needed for the connection as well.

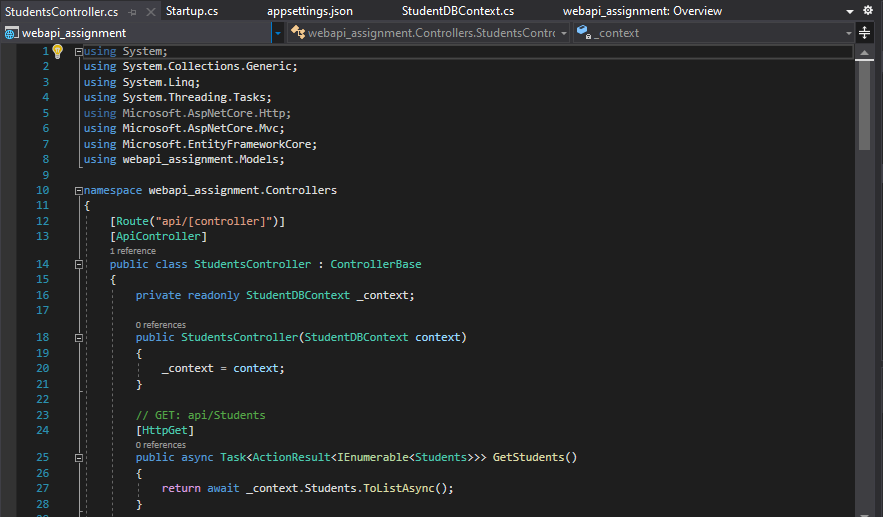


1. After this, a Controller with action methods is scaffolded using entity framework by selecting the model class and the dB context file.

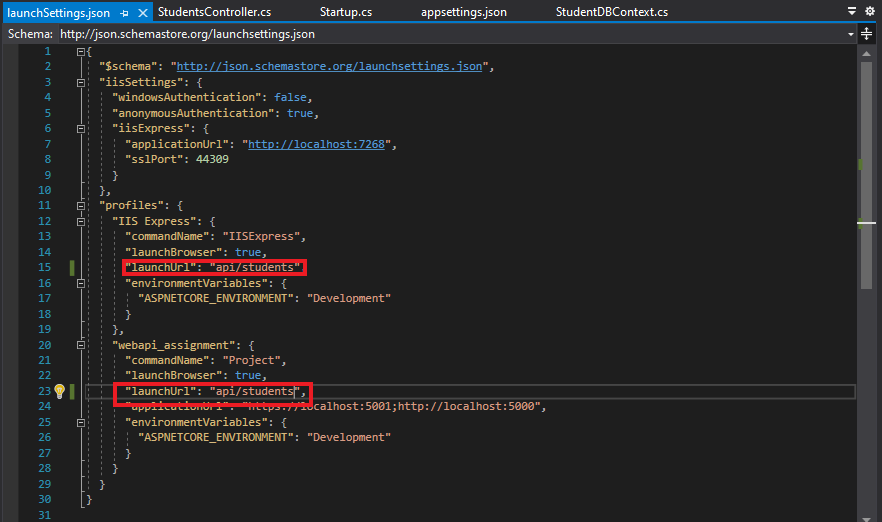
****

****

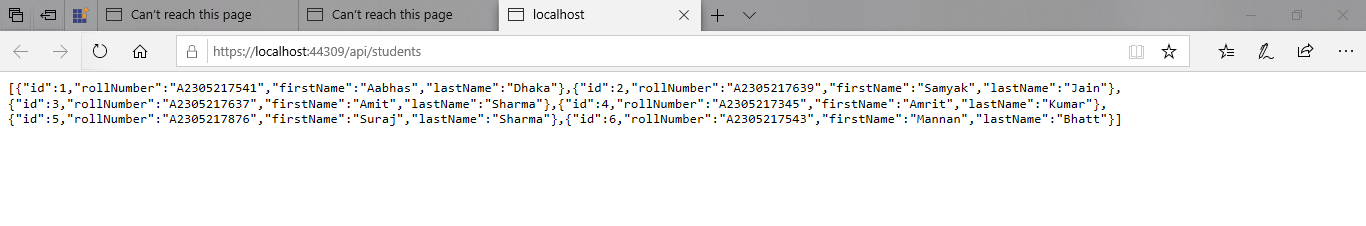
1. Scaffolding provides us with a *StudentsController.cs* file in the controller folder which has all the actions methods such as Get, Post, Get by id, etc.



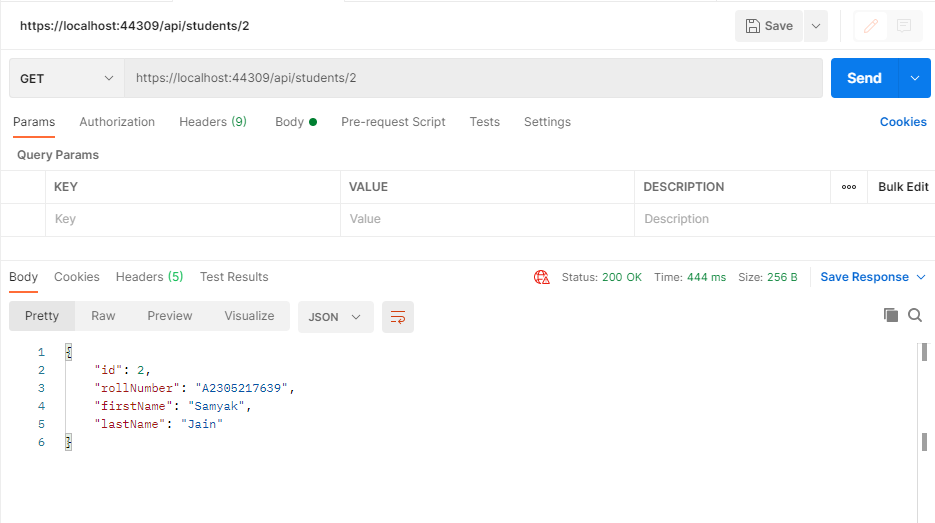
1. Now we need to go to the properties and open the *launchSettings.json file* to change the *launchUrl* of web API.



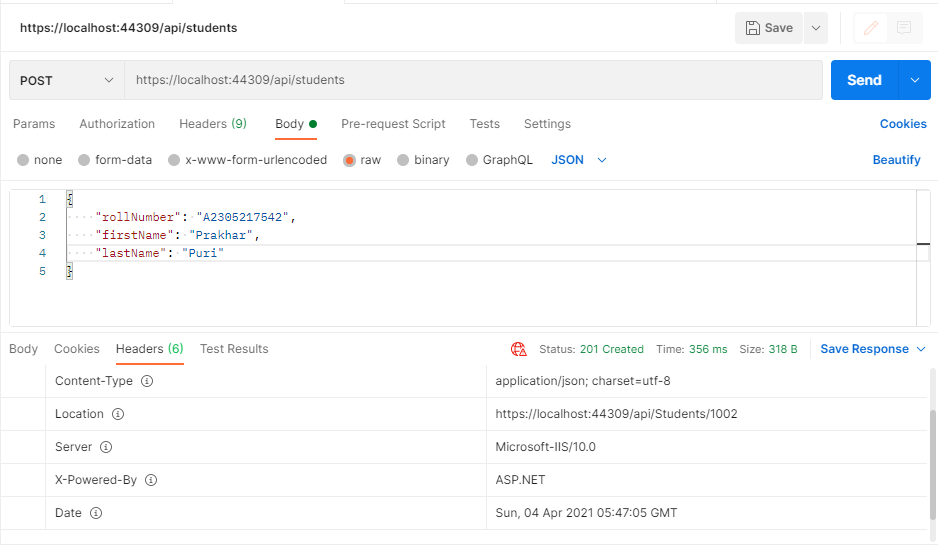
1. On running the application without debugging mode it takes us to the local host and makes the API call automatically because of the previous step.

****

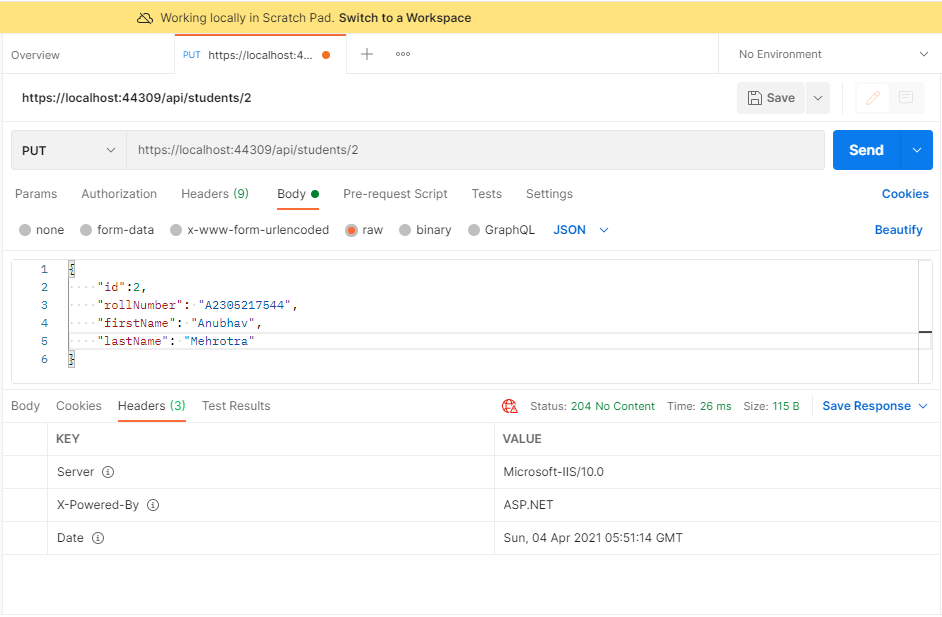
1. **Demonstrating the Get by id method using Postman:**

****

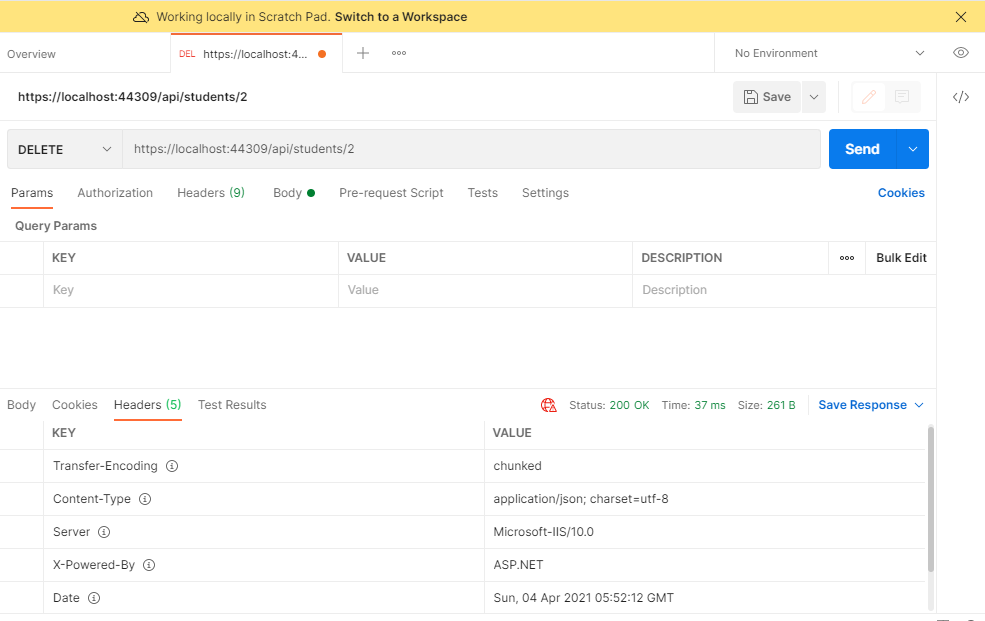
1. **Demonstrating POST method by mentioning data to be posted in the body of http request in the JSON format using Postman:**

****

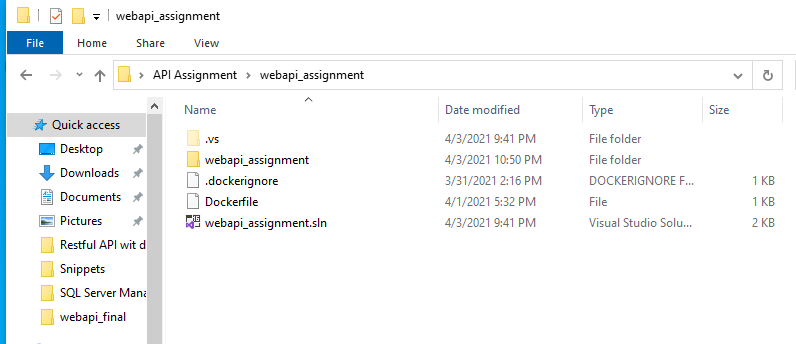
1. **Demonstrating the PUT method by mentioning the data to be updated in the body of http request in the JSON format using Postman:**

****

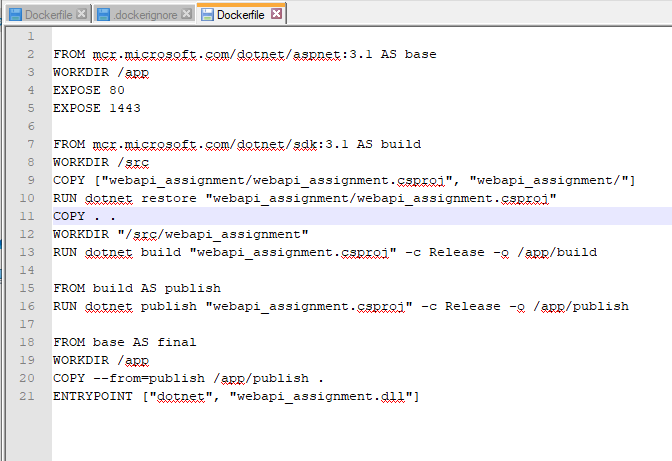
1. **Demonstrating the DELETE by id method using Postman:**

****

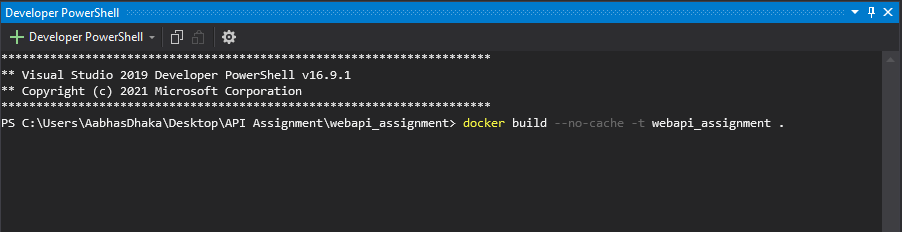
1. Next Step involves dockerizing the .NET Core application. To achieve this first we add two files, namely *Dockerfile* and a *.dockerignore* file.



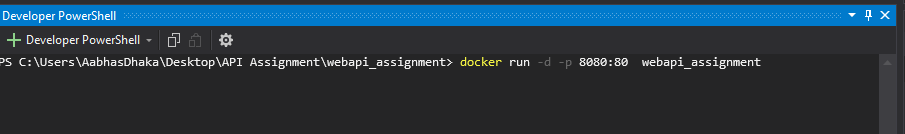
1. With the file in place, we add content to the Dockerfile. It is necessary to expose the port 1433 and port 80. Port 1433 is necessary to expose because using this port our API will be able to communicate with the database which is listening on this port. Port 80 is exposed so that we can communicate with the API using Postman. Dockerfile also includes commands to copy all the project files inside the container, running *dotnet restore* and to publish the application inside the container. It has an entry point which essentially runs the application dll.



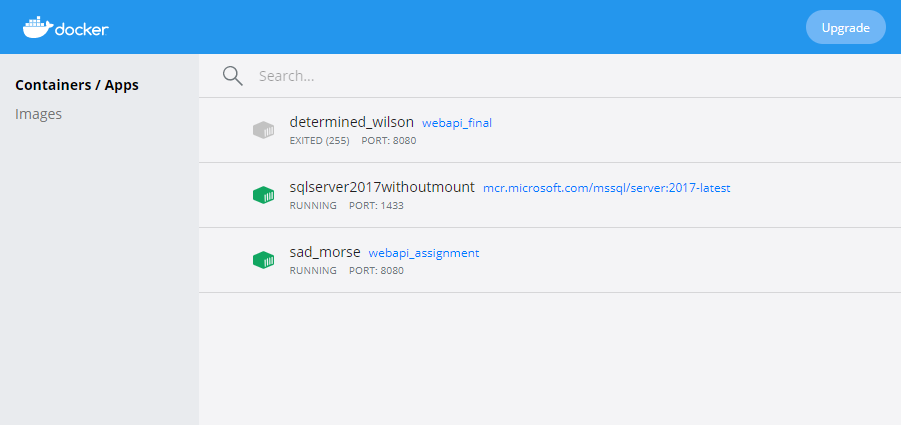
1. Now we use the *docker build* command to build an image of our application inside the terminal.



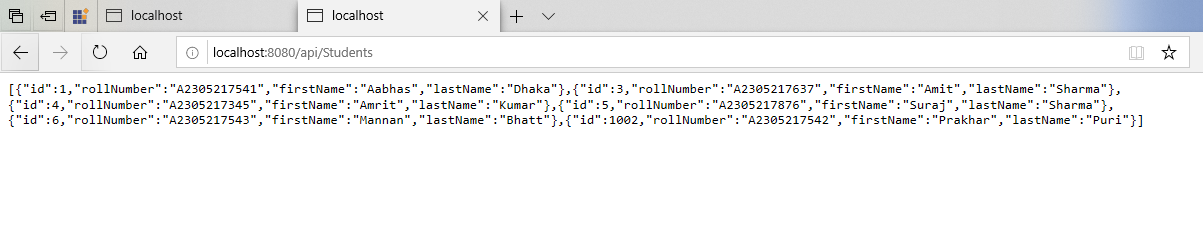
1. After building the image we deploy a container with the image previously built using the *docker run* command. Also, in the command it is necessary to map the port 8080 to port 80 as we had decided to expose port 80 in the Dockerfile.



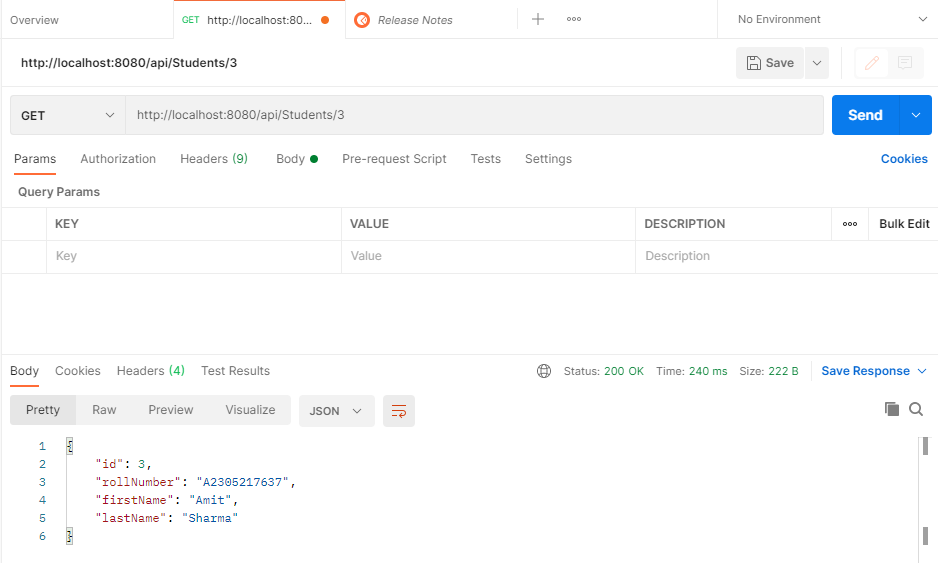
1. As evident from the docker daemon it is seen the both the containers are running on the port 1433 and port 80, respectively.



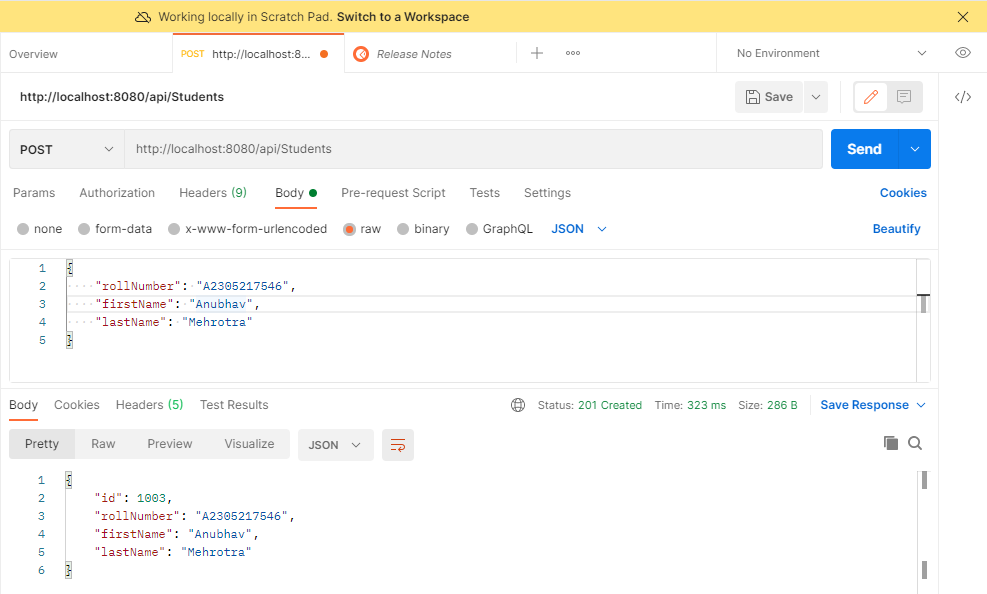
1. After running the local host in the browser with the API call, it is evident that the GET method for the API is working.



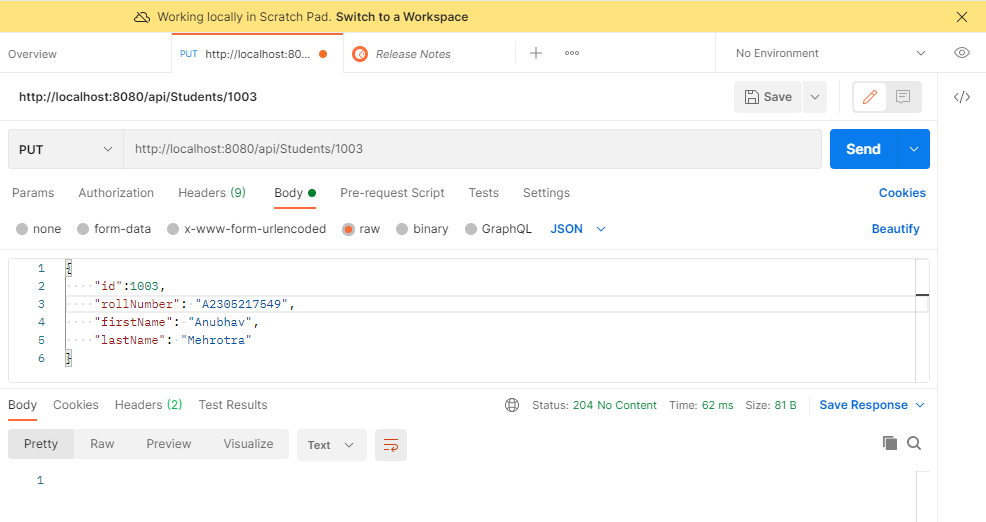
1. **Now, demonstrating the Get by id method using Postman:**

****

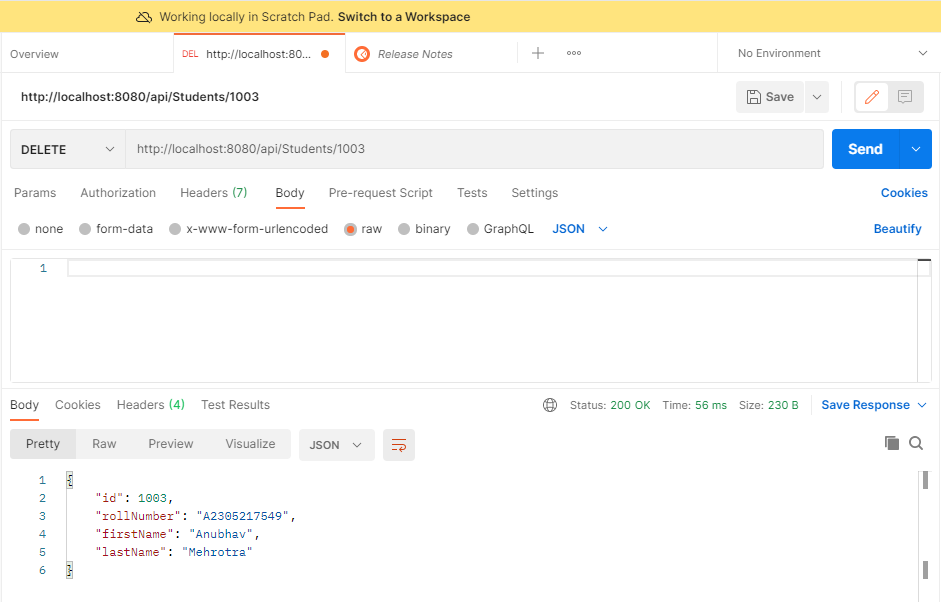
1. **Demonstrating POST method by mentioning data to be posted in the body of http request in the JSON format using Postman:**

****

1. **Demonstrating the PUT method by mentioning the data to be updated in the body of http request in the JSON format using Postman:**

****

1. **Demonstrating the DELETE by id method using Postman:**

****

**Hereby, the assignment has been completed with the ASP.Net Core Web API running inside a container with the database connectivity. The Database itself is running inside a container using Docker.**

**Following are my Learnings from the Assignment:**

1. **Learnt the basics about ASP.Net Core Web API’s.**
2. **Querying the MySQL Database with standard queries and stored procedures.**
3. **Connecting the Database with the ASP.Net application.**
4. **Pulling Container images from various container registries.**
5. **The MVC (Model View Controller) Architecture.**
6. **Using the Entity Framework to scaffold data Models.**
7. **Mapping Ports while deploying images in containers.**