

Programming Skills (Design Patterns and C#)

CSCI-641

Lab 3: An API that interacts with a DB

Documentation

Dipesh Nainani id : dsn1945@rit.edu

1. Overview

This project deals with ASP.net core API's and their interaction with the Database. In this project, Visual Studio 2017 is used to develop a web based API in the ASP.net core framework. The API is then connected to PostgreSQL which stores our data. The application has four entities namely: Library, Book, Author, and Patron. These entities allow the users to perform CRUD operations.

The application will have models, controllers, repositories, startup file and program file. The model contains the variables on which the operations will be performed. The controllers performs the operations such put, post, get, delete by calling the functions from the repositories. The repositories have the definition of these functions and they call an interface and implement the methods of that interface.

2. Tools Used

Visual Studio 2017

PostgreSQL

Postman

3. Setup:

For The Application:

1) Install Visual Studio 2017

2) Create a Web API

File -> New -> Project -> .NET Core -> ASP.NET Core Web Application(.Net Core) (here you can give a name for the project)-> Web API

3) The Startup.cs, Program.cs files and the Controllers folder will be created by default.

4) Create a folder for model

Right Click on project name -> Add -> New Folder -> name it Models

5) Create a folder for the repositories

Right Click on project name -> Add -> New Folder -> name it Services

6) Install the required packages:

Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution -> Browse -> search for dapper, npgsql and Swashbuckle.AspNetCore (for swagger)

7) In the controllers folder, create individual controllers for the entities

Right Click -> Add -> class -> Web API Controller Class

(Create controllers for every entity: Library, Patron, Book, Author)

8) In the services folder, create the individual classes and individual interfaces for the each of the entities.

For Interface,

Right Click on the services folder -> Add -> Class -> Interface

For Repositories,

Right Click on the services folder -> Add -> Class -> Class

(Name them according to the entities)

9) In the model folder, create a model for the entities, you can also create different models for each entity.

Right Click on the Model folder -> Add -> New Folder -> name it on any entity

For the database:

1) Install the latest version of PostgreSQL

2) Setup the default port and password for the database.

3) Create a database:

Right Click on Databases option -> New Database -> give a name(should be in small letters)

4) Create tables:

Right Click on the created database -> CREATE SCRIPT(if you want to execute your own scripts and) -> Run them

OR

Click on the database -> Click on Schemas -> public -> Right click on the Tables option and create tables for each entity and provide the columns.

5) Add values in the table accordingly.

For Viewing the Results:

1) Install the latest version of Postman.

4. Design:

(a) For the Models:

In the models, class for each entity is defined. The classes will declare the variables on which the operation will be performed. The model will be used by the controllers.

(b) For the Controllers: The methods in the controllers will be created by default when they are made. They will have the methods Get, Put, Post and Delete, which will call the methods from the repositories and execute the functions.

(c) For the Repositories: These implement the methods in the interface and execute SQL queries in them. Database connection will be opened here with the help of connectionstring passed in the appsettings.json file. The queries will be performed on the tables of the database passed in the connectionstring.

(d) For the Interface: Interface made for each of the entities include the following methods: add, getbyid, getall, delete and update. These methods will be implemented by the repositories class further.

(e) For database: To connect to the PostgreSQL, you have to first write lines of the code in the Startup.cs file. The ConfigureServices and Configure Method will include code to configure the services of the database in the project.

The appsettings.json file includes, the database info that is the name and the connectionstring which has the userid, password, host, databasename.

(f) For Swagger: The documentation of API is done in the Swagger. To view these, you have to add code in the ConfigureServices and Configure Method present in the Startup.cs file.

After running the project, the default browser will get opened. The address bar will have the url named localhost: <http://localhost:2496/api/values>. In-place of that url, write <http://localhost:2496/swagger>.

Then you will get to view the whole API and perform the operations simultaneously. Each entity will be present in the browser.

(g) For Postman: For viewing the results, postman can also be used. When the project starts building, the default browser opens up which has an address.

Copy that address and paste the address in the url bar of postman. You can then simultaneously perform the operations.

5. **Results:** The methods in the repositories handle the execution of queries. The book repository helps in adding multiple books to one single author. In the add function of author multiple books can be added to one single author.

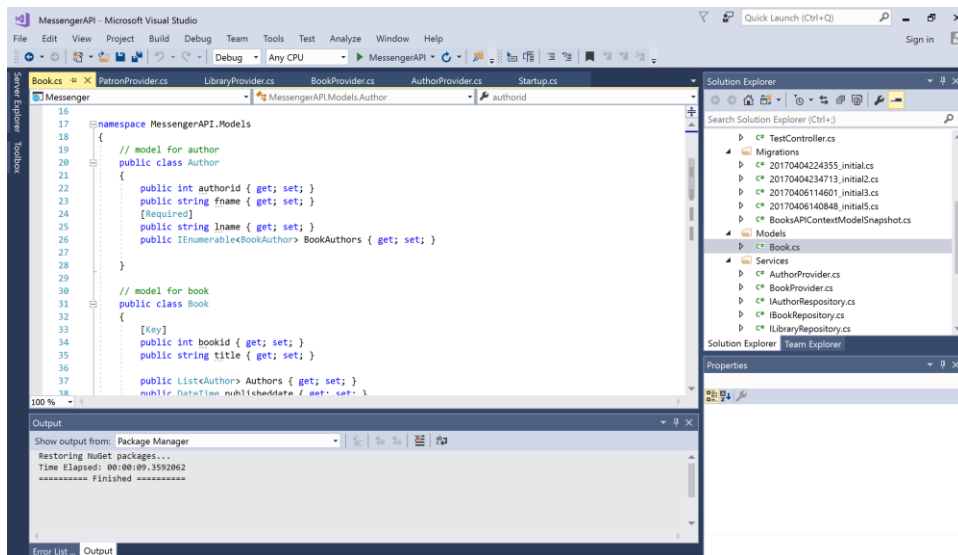
Simultaneously, multiple authors can be added to one book. The add function of the book repository has a list of the names of the authors stored and gets added to that particular book.

The CRUD operations are implemented for each of the entity.

Simultaneously, other operations are also handled in the project.

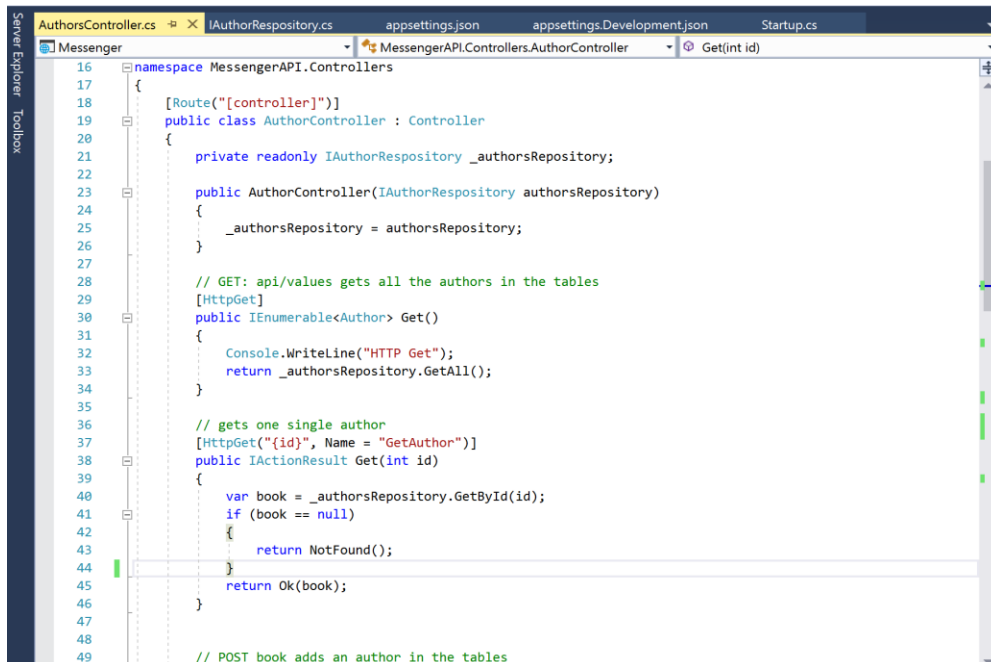
6. Screenshots

A) For model:



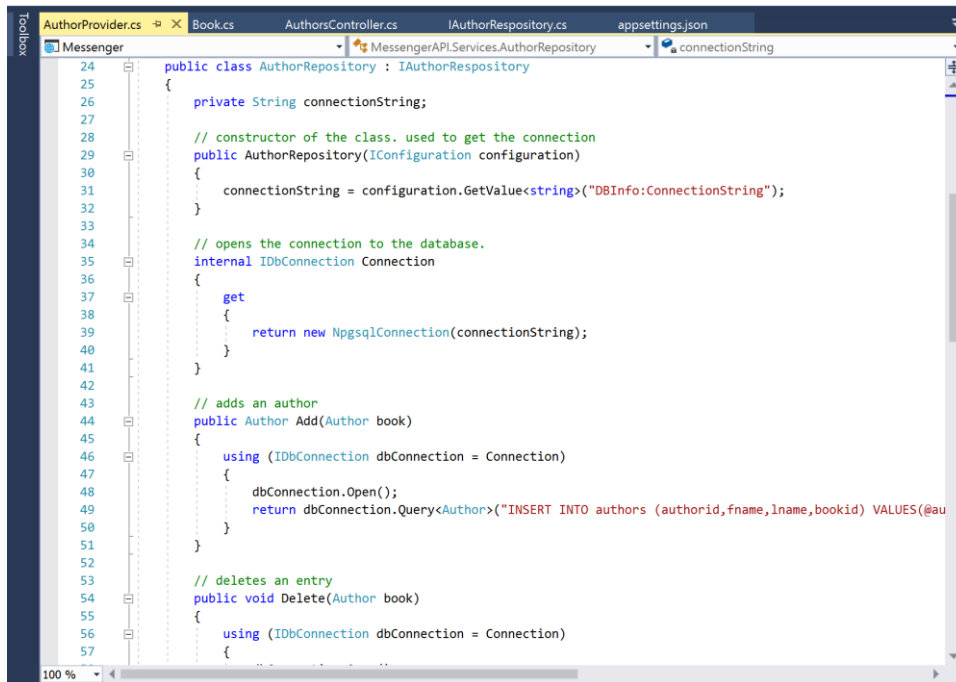
The screenshot shows the model for the author. The variables are declared in the class which will be used by the controllers. Similarly, for each entity the class will be created with their variables.

B) For Controller:



The screenshot shows the controller for the author. Similarly, the other entities will have their controllers defined. It by default has the methods Get, Post, Put and Delete.

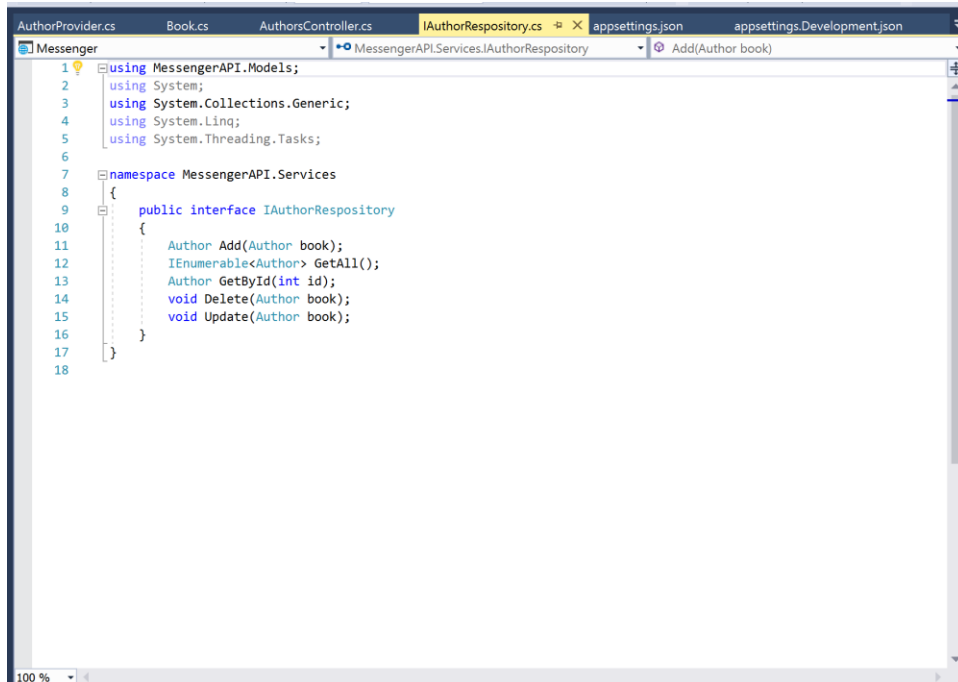
C) For Repository:



```
24 public class AuthorRepository : IAuthorRepository
25 {
26     private String connectionString;
27
28     // constructor of the class. used to get the connection
29     public AuthorRepository(IConfiguration configuration)
30     {
31         connectionString = configuration.GetValue<string>("DBInfo:ConnectionString");
32     }
33
34     // opens the connection to the database.
35     internal IDbConnection Connection
36     {
37         get
38         {
39             return new NpgsqlConnection(connectionString);
40         }
41     }
42
43     // adds an author
44     public Author Add(Author book)
45     {
46         using (IDbConnection dbConnection = Connection)
47         {
48             dbConnection.Open();
49             return dbConnection.Query<Author>("INSERT INTO authors (authorid,fname,lname,bookid) VALUES(@au
50         }
51     }
52
53     // deletes an entry
54     public void Delete(Author book)
55     {
56         using (IDbConnection dbConnection = Connection)
57     {
```

This class implements the methods in the interface. In this screenshot, AuthorRepository implements the interface IAuthorRepository. In this, the methods execute SQL Queries on the database and open the connection to the database from the connectionString.

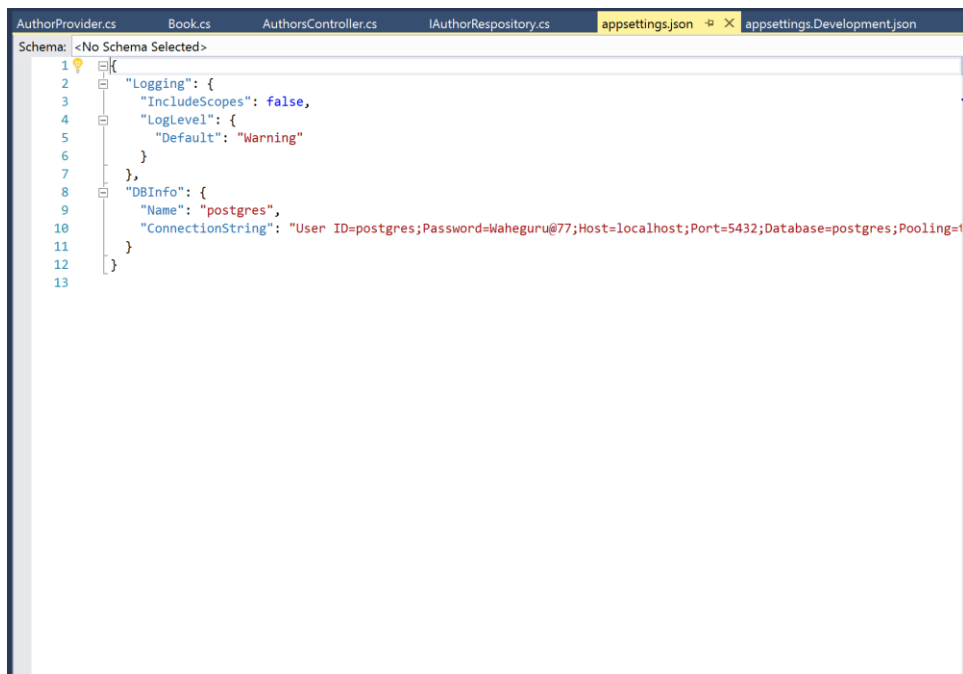
D) For Interface:



```
1 using MessengerAPI.Models;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Threading.Tasks;
6
7 namespace MessengerAPI.Services
8 {
9     public interface IAuthorRespository
10     {
11         Author Add(Author book);
12         IEnumerable<Author> GetAll();
13         Author GetById(int id);
14         void Delete(Author book);
15         void Update(Author book);
16     }
17 }
18
```

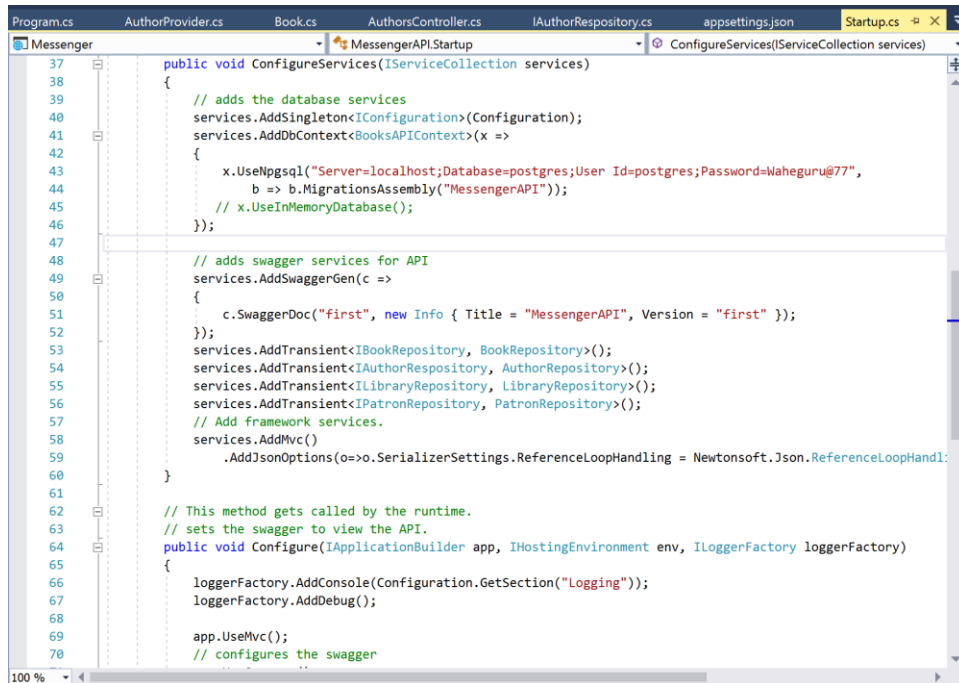
This contains the methods which the interface will contain. Similarly, the other entities can have their own interface with their own methods.

E) For Database and Swagger:



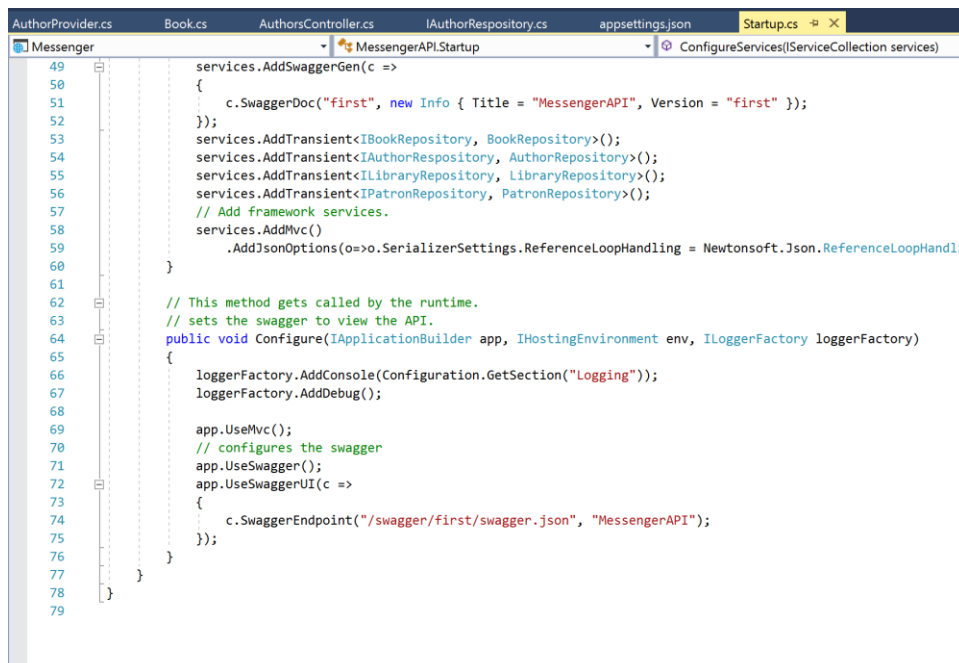
```
1 {
2   "Logging": {
3     "IncludeScopes": false,
4     "LogLevel": {
5       "Default": "Warning"
6     }
7   },
8   "DBInfo": {
9     "Name": "postgres",
10    "ConnectionString": "User ID=postgres;Password=Waheguru@77;Host=localhost;Port=5432;Database=postgres;Pooling=1"
11  }
12 }
13
```

This is the appsettings.json file which has the connectionstring of the database and contains all its information.



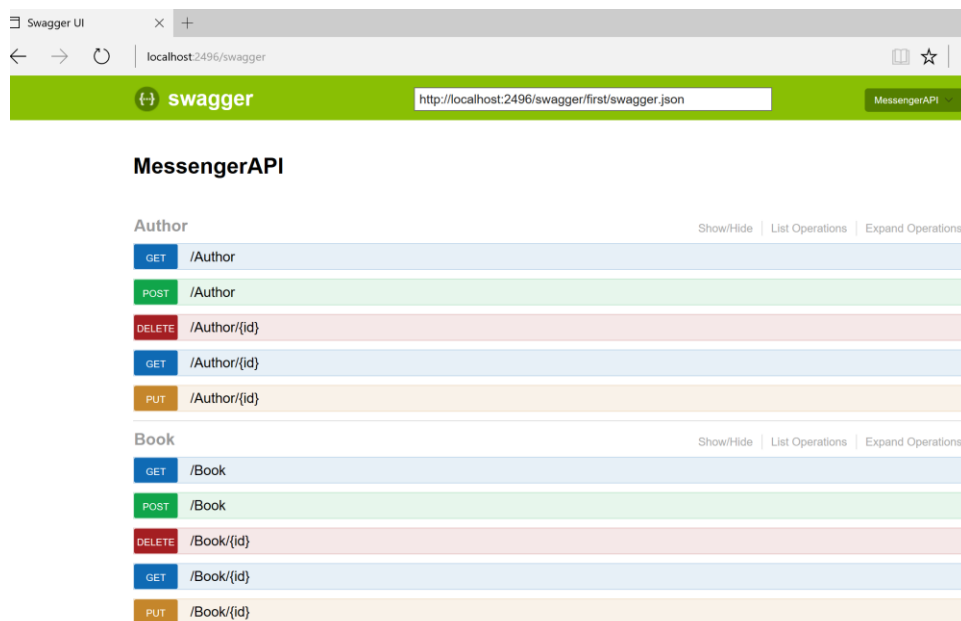
```
Program.cs AuthorProvider.cs Book.cs AuthorsController.cs IAuthorRepository.cs appsettings.json Startup.cs
Messenger MessengerAPI.Startup ConfigureServices(IServiceCollection services)
37 public void ConfigureServices(IServiceCollection services)
38 {
39     // adds the database services
40     services.AddSingleton<IConfiguration>(Configuration);
41     services.AddDbContext<BooksAPIContext>(x =>
42     {
43         x.UseNpgsql("Server=localhost;Database=postgres;User Id=postgres;Password=Waheguru@77",
44         b => b.MigrationsAssembly("MessengerAPI"));
45         // x.UseInMemoryDatabase();
46     });
47
48     // adds swagger services for API
49     services.AddSwaggerGen(c =>
50     {
51         c.SwaggerDoc("first", new Info { Title = "MessengerAPI", Version = "first" });
52     });
53     services.AddTransient<IBookRepository, BookRepository>();
54     services.AddTransient<IAuthorRepository, AuthorRepository>();
55     services.AddTransient<ILibraryRepository, LibraryRepository>();
56     services.AddTransient<IPatronRepository, PatronRepository>();
57     // Add framework services.
58     services.AddMvc()
59     .AddJsonOptions(o=>o.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandl
60
61 }
62
63 // This method gets called by the runtime.
64 // sets the swagger to view the API.
65 public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
66 {
67     loggerFactory.AddConsole(Configuration.GetSection("Logging"));
68     loggerFactory.AddDebug();
69
70     app.UseMvc();
71     // configures the swagger
72     app.UseSwagger();
73     app.UseSwaggerUI(c =>
74     {
75         c.SwaggerEndpoint("/swagger/first/swagger.json", "MessengerAPI");
76     });
77 }
78
79
```

This is the Startup.cs file which contains the code to setup the database and swagger. This is the configureservice method.



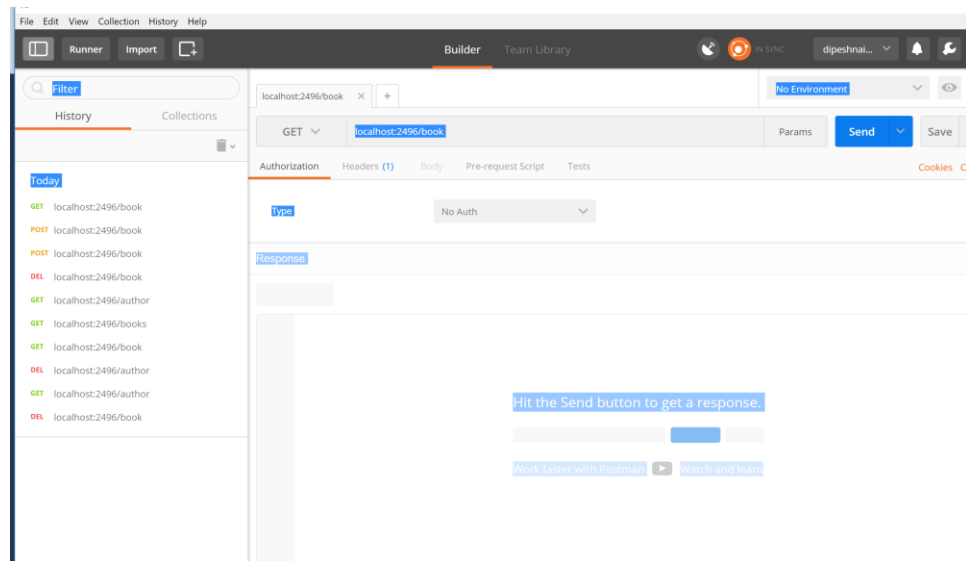
```
AuthorProvider.cs Book.cs AuthorsController.cs IAuthorRepository.cs appsettings.json Startup.cs
Messenger MessengerAPI.Startup ConfigureServices(IServiceCollection services)
49 services.AddSwaggerGen(c =>
50 {
51     c.SwaggerDoc("first", new Info { Title = "MessengerAPI", Version = "first" });
52 });
53 services.AddTransient<IBookRepository, BookRepository>();
54 services.AddTransient<IAuthorRepository, AuthorRepository>();
55 services.AddTransient<ILibraryRepository, LibraryRepository>();
56 services.AddTransient<IPatronRepository, PatronRepository>();
57 // Add framework services.
58 services.AddMvc()
59 .AddJsonOptions(o=>o.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandl
60
61 }
62
63 // This method gets called by the runtime.
64 // sets the swagger to view the API.
65 public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
66 {
67     loggerFactory.AddConsole(Configuration.GetSection("Logging"));
68     loggerFactory.AddDebug();
69
70     app.UseMvc();
71     // configures the swagger
72     app.UseSwagger();
73     app.UseSwaggerUI(c =>
74     {
75         c.SwaggerEndpoint("/swagger/first/swagger.json", "MessengerAPI");
76     });
77 }
78
79
```


This is the configure method which also contains code to setup the database and swagger.



This is how swagger runs. It performs all the operations for all the entities present in the application. Also, the URL has <http://localhost:2496/swagger> in place of <http://localhost:2496/api/values>.

F) For Postman:



This is how the postman executes the application. The url of the project needs to be specified in url bar and then all the operations can be performed.