CIS 466

Systems Development ASP.Net Project

Summer 2018

Developer Documentation

Kaiba Redux Team:

**Kyle Tejada**

**Imran Parvez** **Steve Mori**

**Jose Rodriguez**

**Nathan Wong**

**Vincent Luu**

**Table of Contents**

[**1 INTRODUCTION** 3](#_Toc519832199)

[**2 PREREQUISITES** 3](#_Toc519832200)

[**2.1 Install Visual Studio, The Existing API, and .Net Core 2.1** 3](#_Toc519832201)

[**2.2 Install SQL Server 2017 Standard Edition** 4](#_Toc519832202)

[**3 CONFIGURATIONS** 11](#_Toc519832203)

[**3.1 Establishing a Connection from the KaibaReduxAPI to SQL Server** 11](#_Toc519832204)

[**3.2 Configuring the KaibaRedux Database in SQL Server** 12](#_Toc519832205)

[**3.3 Reset the KaibaRedux Database in SQL Server** 14](#_Toc519832206)

[**4 APPLICATION** 15](#_Toc519832207)

[**4.1 Running the Kaiba Redux API** 15](#_Toc519832208)

[**4.2 Create a New Menu in the Kaiba Redux API** 16](#_Toc519832209)

[**4.3 Read a New Menu in the API** 16](#_Toc519832210)

[**4.4 Update and Editing Existing Items** 17](#_Toc519832211)

[**4.5 Delete Items from the Application Program Interface** 18](#_Toc519832212)

# **1 INTRODUCTION**

## **1.1 What is a Web API?**

An API (Application Programming Interface) is a system of functions and procedures to use someone else’s prewritten code. For example, Google has an API that allows developers to access many Google Maps features within their own applications. A web API is simply one that is accessed over an internet connection using URLs. This guide describes the creation and implementation of an API for displaying and editing a restaurant menu.

This application has three tiers. The three layers are the database layer, application layer, and the client layer. The database layer consists of a Microsoft SQL Server database instance. This layer stores all the information. The application or business logic layer consists of several ASP.NET classes written in C#. It queries the database in order to create, retrieve, update or delete (CRUD) information in the database. It also provides a web API that can take or receive JSON data from the client. The client layer consists of some HTML pages and several dynamic JavaScript (JS) functions.

This project is powered by ASP.NET. uses Microsoft’s Active Server Pages (ASP) technology used to design dynamic webpages and provides all the services that is required to build enterprises-level class server web applications. ASP.NET is part of a .NET framework in which allows access within the classes of the .NET environment. Applications can be coded in various languages compatible within Common Language Runtime (CLR) which enhances the further development of ASP.NET applications from CLR, type safety, and inheritance.

## **1.2 Recommended Prior Experience**

This is a fairly complicated project that includes many different technologies and is written in several different programming languages.

Familiarity with the following is **strongly** recommended:

1. Multi-Tier Applications: specifically the Model, View, Controller (MVC) pattern
2. Databases and SQL queries
3. Visual Studio, the IDE used
4. The C# Programming Language,
   1. Or the Java Programming Language (very similar in structure and syntax to C#)
5. HTML and web page creation
6. JavaScript, the scripting language that runs within web pages
   1. JQuery, a common JS library that simplifies many common JS actions

# **2 PREREQUISITES**

## **2.1 Install Visual Studio, The Existing API, and .Net Core 2.1**

This project is intended for Windows 7+ machines only. For MacOS and Linux systems certain things may not work and might require different installation directions. We cannot guarantee that this project will work or even install on non-Windows systems.

Install Visual Studio here:

[https://www.visualstudio.com/thank-you-downloading-visual-studio/?sku=community&rel=15#](https://www.visualstudio.com/thank-you-downloading-visual-studio/?sku=community&rel=15)

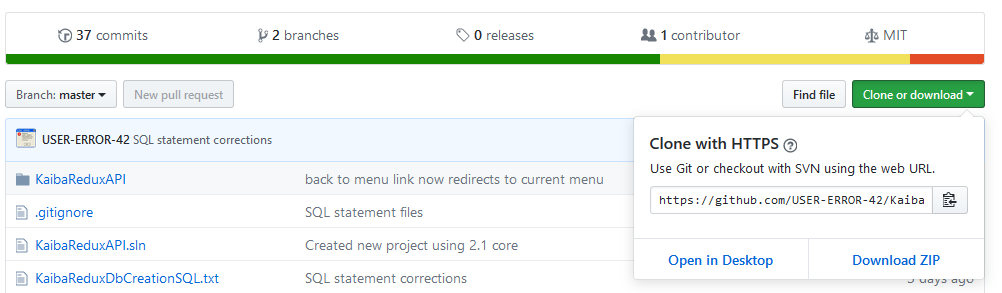
Select **.NET Core cross-platform development** installation and click **Install**.

Add the existing web application program interface (API) from the **GitHub** repository named **KaibaReduxAPI** in the following steps:

1. Visit <https://github.com/USER-ERROR-42/KaibaReduxAPI>
2. Click on “**Clone or download”** and choose a method in the picture below
3. “**Open in Desktop”** if GitHub Desktop is preferred.

Install Guide GitHub Desktop:

<https://help.github.com/desktop/guides/contributing-to-projects/cloning-a-repository-from-github-to-github-desktop/> or **“Download ZIP”**



1. After the repository has been cloned or unzipped to a preferred location on your Windows PC open **“KaibaReduxAPI.sln”** to build the initial project in Visual Studio. An option to install additional extensions may be required. **Accept additional installs if required.**
2. **Install .NET Core 2.1 SDK or Later**

Download and install .NET SDK (Tested Ver. 2.1.302)

<https://www.microsoft.com/net/download/thank-you/dotnet-sdk-2.1.302-windows-x64-installer>

**\*Additional information about ASP.Net Core 2.1 and guide:**

<https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1>

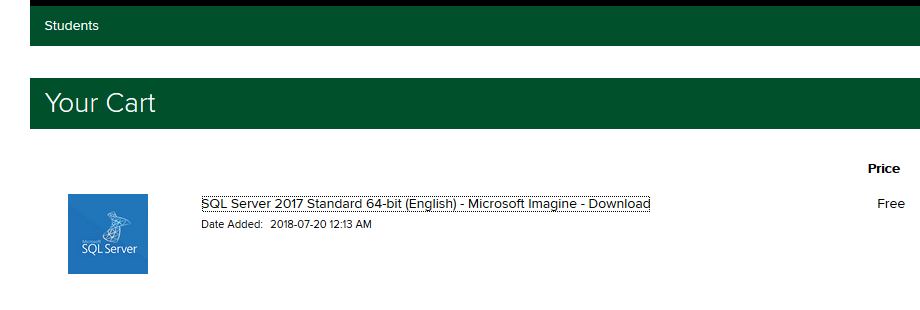
## **2.2 Install SQL Server 2017 Standard Edition**

Install SQL Server 2017 Standard Edition 32/64 bit, here:

<https://e5.onthehub.com/WebStore/Security/Signin.aspx?ws=330817f9-cb9b-e011-969d-0030487d8897&vsro=8&rurl=%2fWebStore%2fProductsByMajorVersionList.aspx%3fws%3d330817f9-cb9b-e011-969d-0030487d8897%26vsro%3d8>

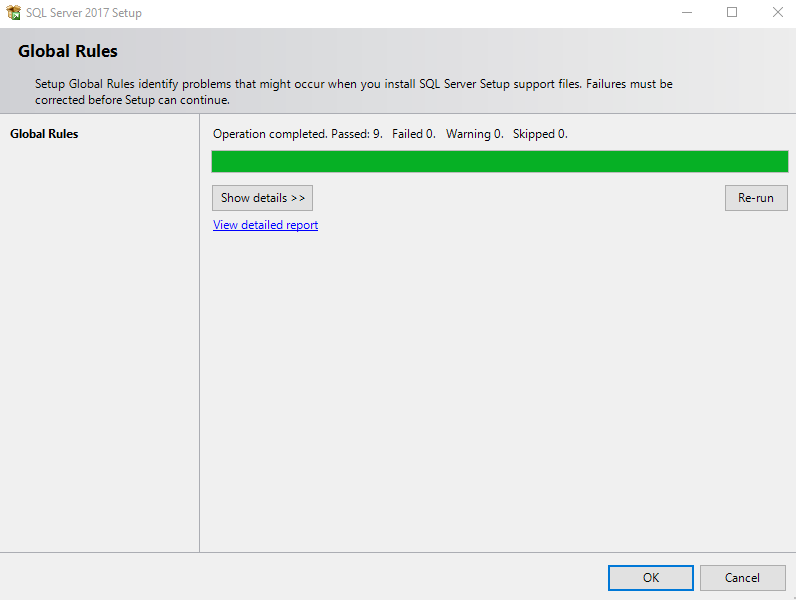
Or Free Developer Edition SQL Server 2017: <https://www.microsoft.com/en-ca/sql-server/sql-server-downloads>

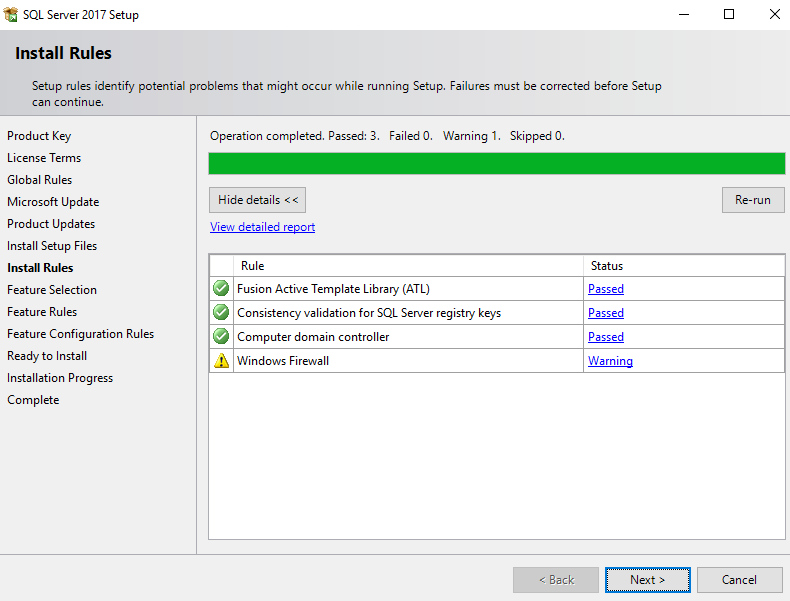
A CPP Student account will be required to download a free licensed copy

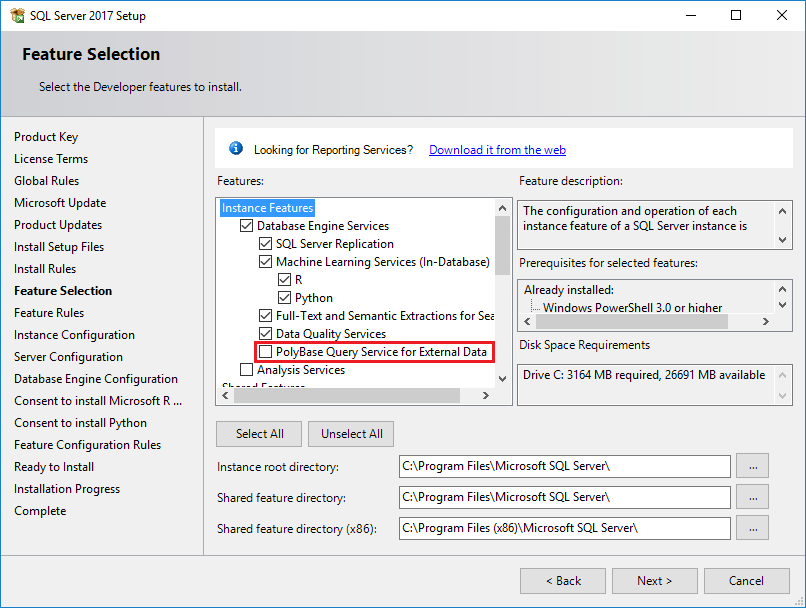


1. Open the **setup.exe** file in the parent directory of the DVD (en\_sql\_server\_2017\_standard\_x64\_dvd\_11294407.iso file).
2. Run **System Configuration Checker** for compatibility.

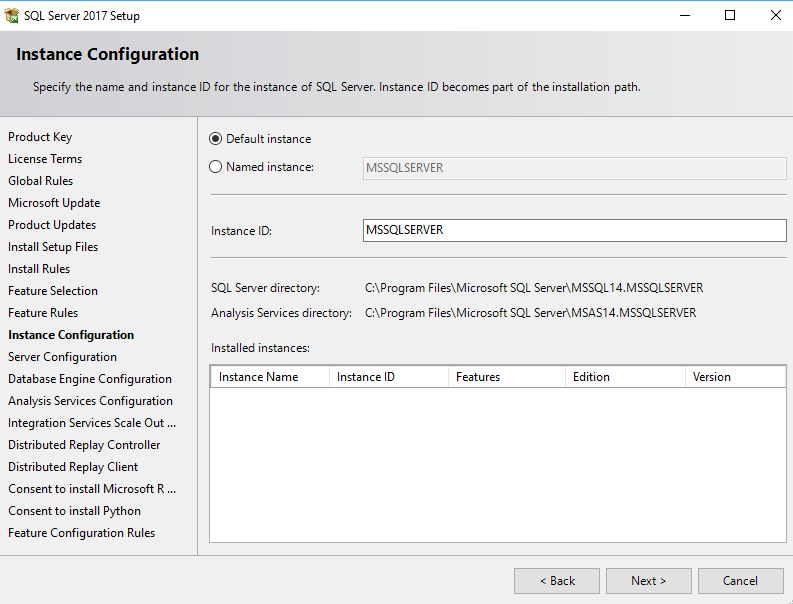




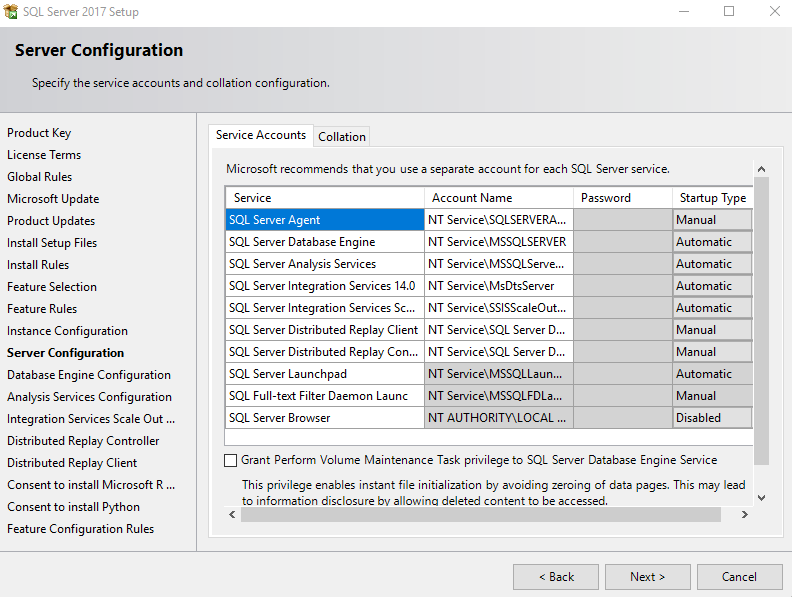
1. Click **“Installation”** on the left select **“New SQL Server stand-alone installation or add features to an existing installation”** if you receive an ERROR about SQL Server not able to update ignore and select “**Next”** 
2. In the **Feature Selection** select everything EXCEPT **“PolyBase Query Service for External Data** (only runs on Oracle JRE 8)**”**



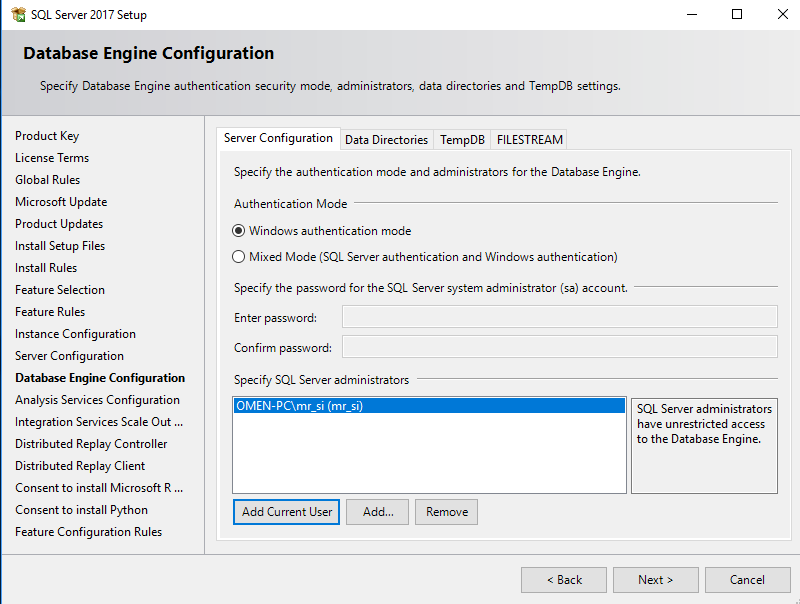
1. Install Java SE Runtime Environment 10.0.2 **Agree** to the terms and download at: <http://www.oracle.com/technetwork/java/javase/downloads/jre10-downloads-4417026.html>
2. Select **“Default Instance”** click on next



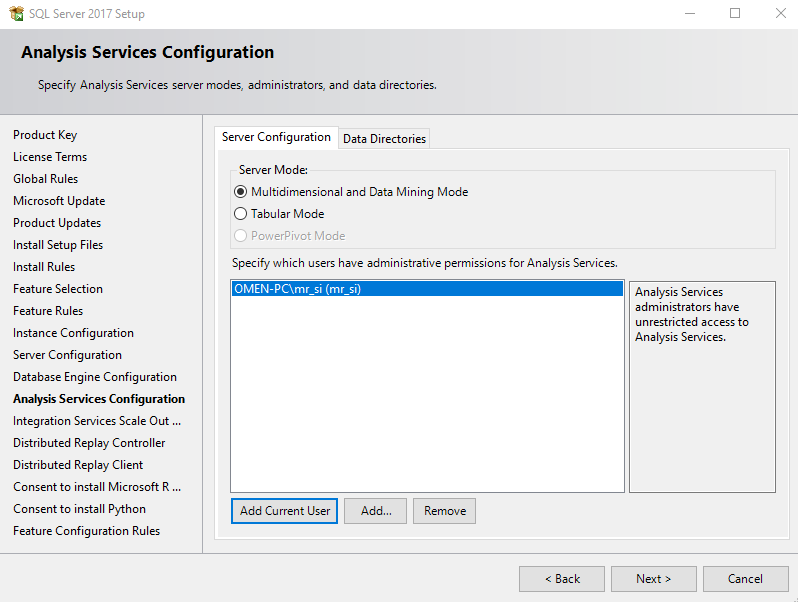
1. The screen below provides a summary of accounts that will be configured. Hit **Next**



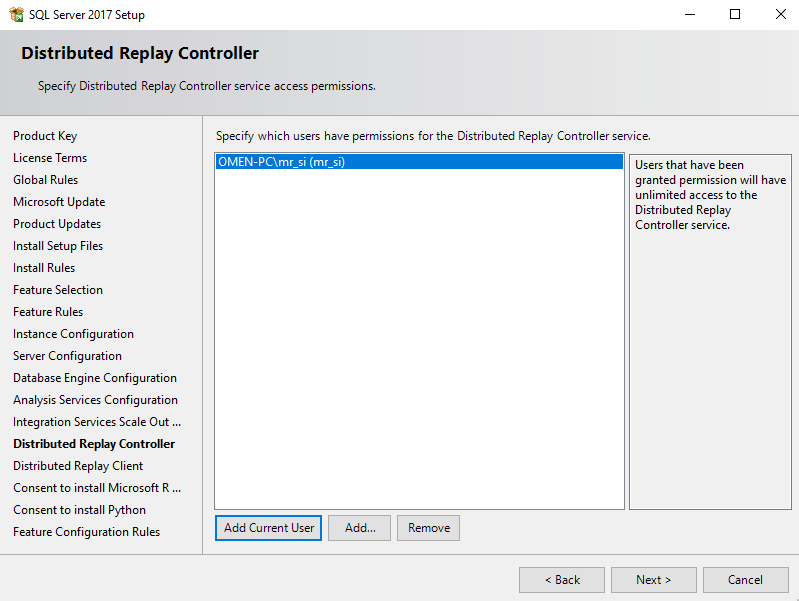
1. Make sure **“Windows authentication mode”** is selected. Click **“Add Current User”** and **Next**



1. Make sure **“Multidimensional and Data Mining Mode”** is clicked in **“Analysis Services Configuration”** and click **“Add Current User”** again. Hit **Next**



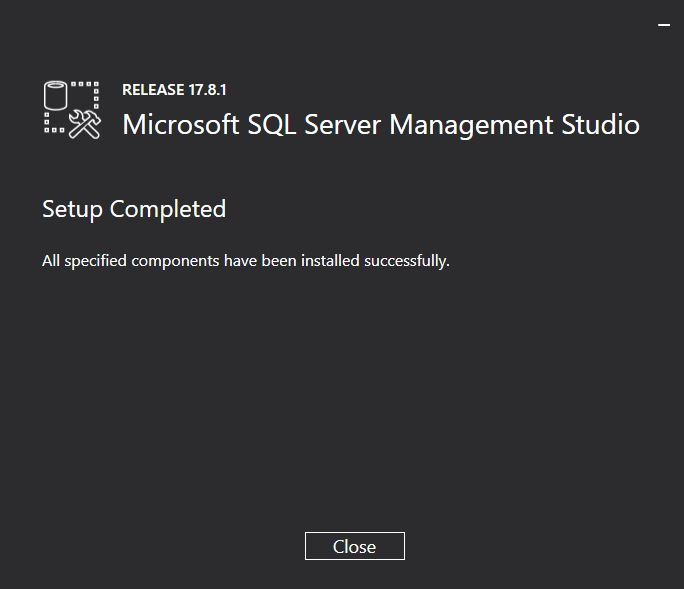
1. Click **Next (Controller names and end nodes may be blank)** until you reach **“Distributed Replay Controller”**
2. click **“Add Current User”** again **Agree** to terms and wait for the install to complete over several minutes.



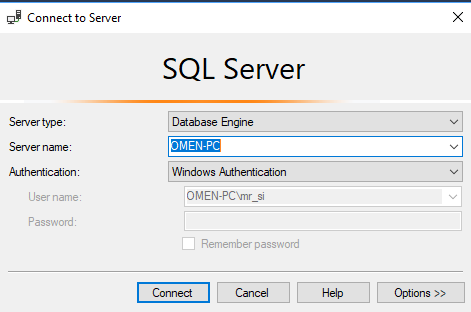
1. Download SQL Server Management Studio 17.8.1 or Later here:

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>

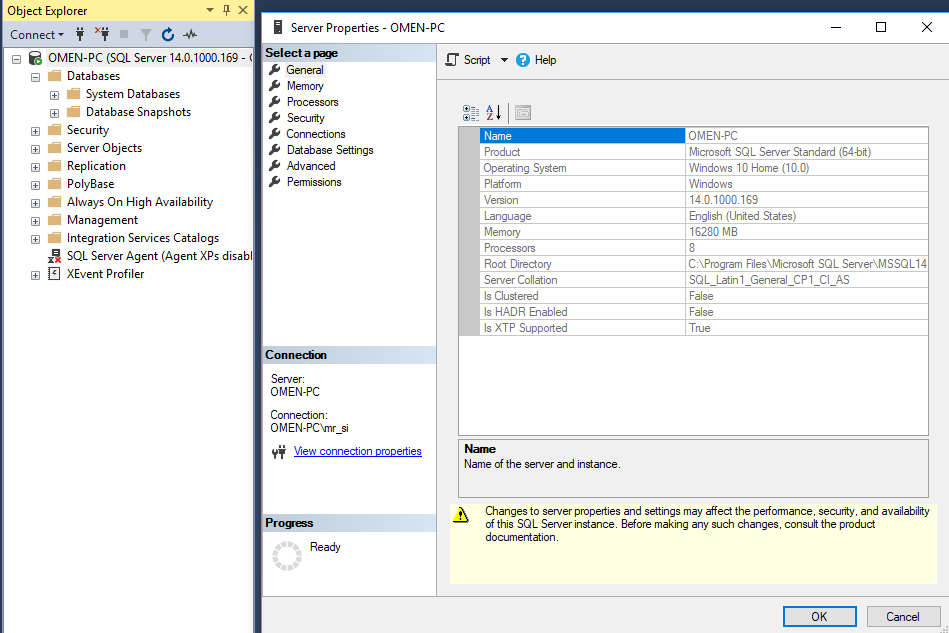
The same link is also provided in the SQL Server initial setup screen. Install. Done.



1. From the Start Menu open **“Microsoft SQL Server Management Studio”** and make note of your **“Server name”** click **“Connect”**



1. **Right clicking** on your Server and clicking **“Properties”** displaysyour server name as well. This is a crucial step and is covered in the next section.



## **2.3 Install POSTMAN**

Download and install POSTMAN here:

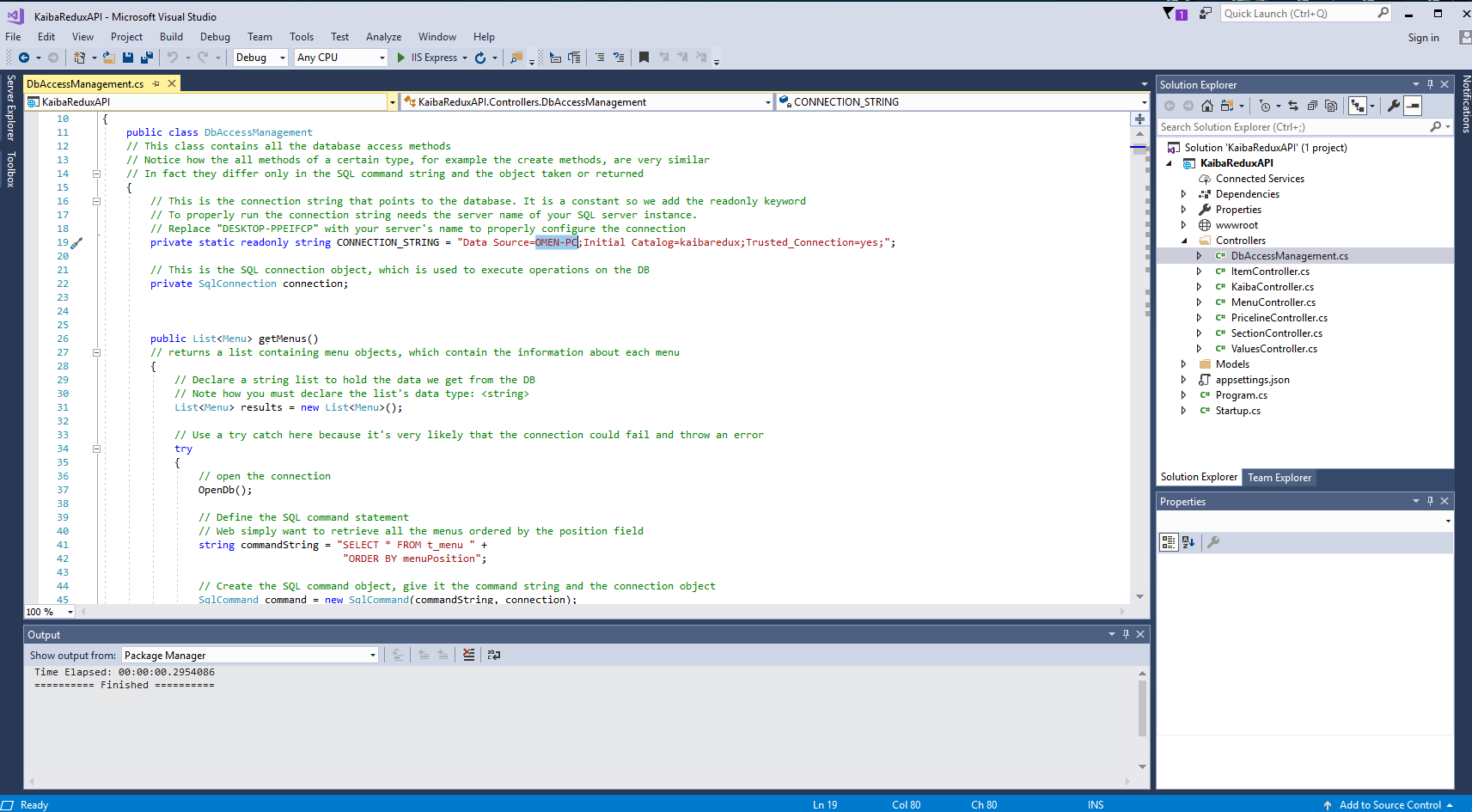
<https://www.getpostman.com/>

# **3 CONFIGURATIONS**

## **3.1 Establishing a Connection from the KaibaReduxAPI to SQL Server**

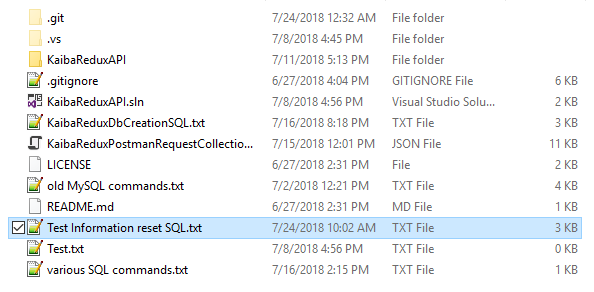
This step is **CRUCIAL** to have proper communication to your database which is dependent to your **Server name (Desktop Name)**

1. Refer to **section 2.1 Step 4**
2. Once Visual Studio has opened the **“KaibaReduxAPI.sln”** file go to the **“Solution Explorer”** taskbar on the right and open the **“Controllers”** folder
3. Open the **“DbAccessManagement.cs”** C# file and add your **Server name** from **section 2.2 Step 14**.
4. Edit after **“Data Source= “** and click the save icon on the top left

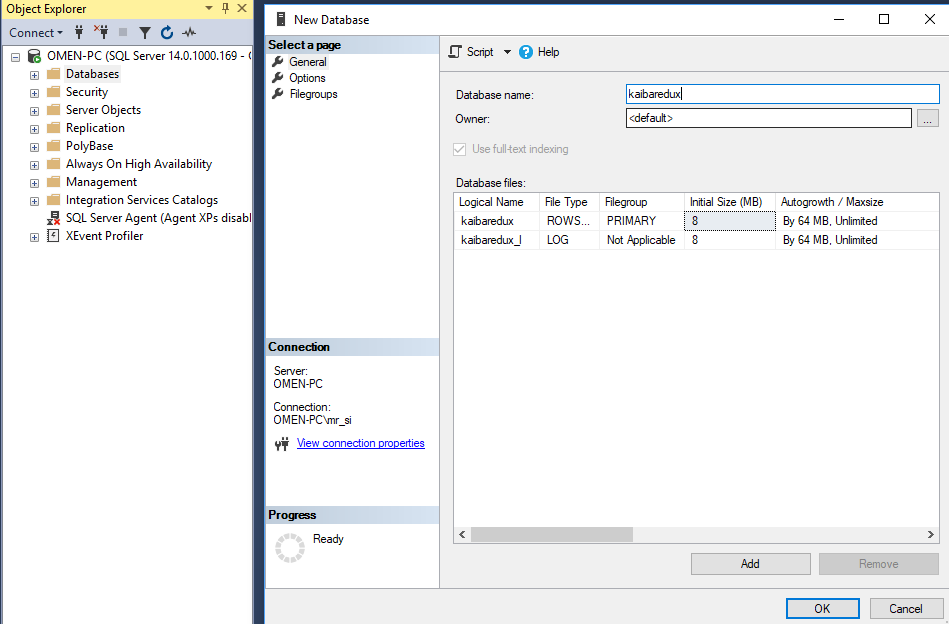


## **3.2 Configuring the KaibaRedux Database in SQL Server**

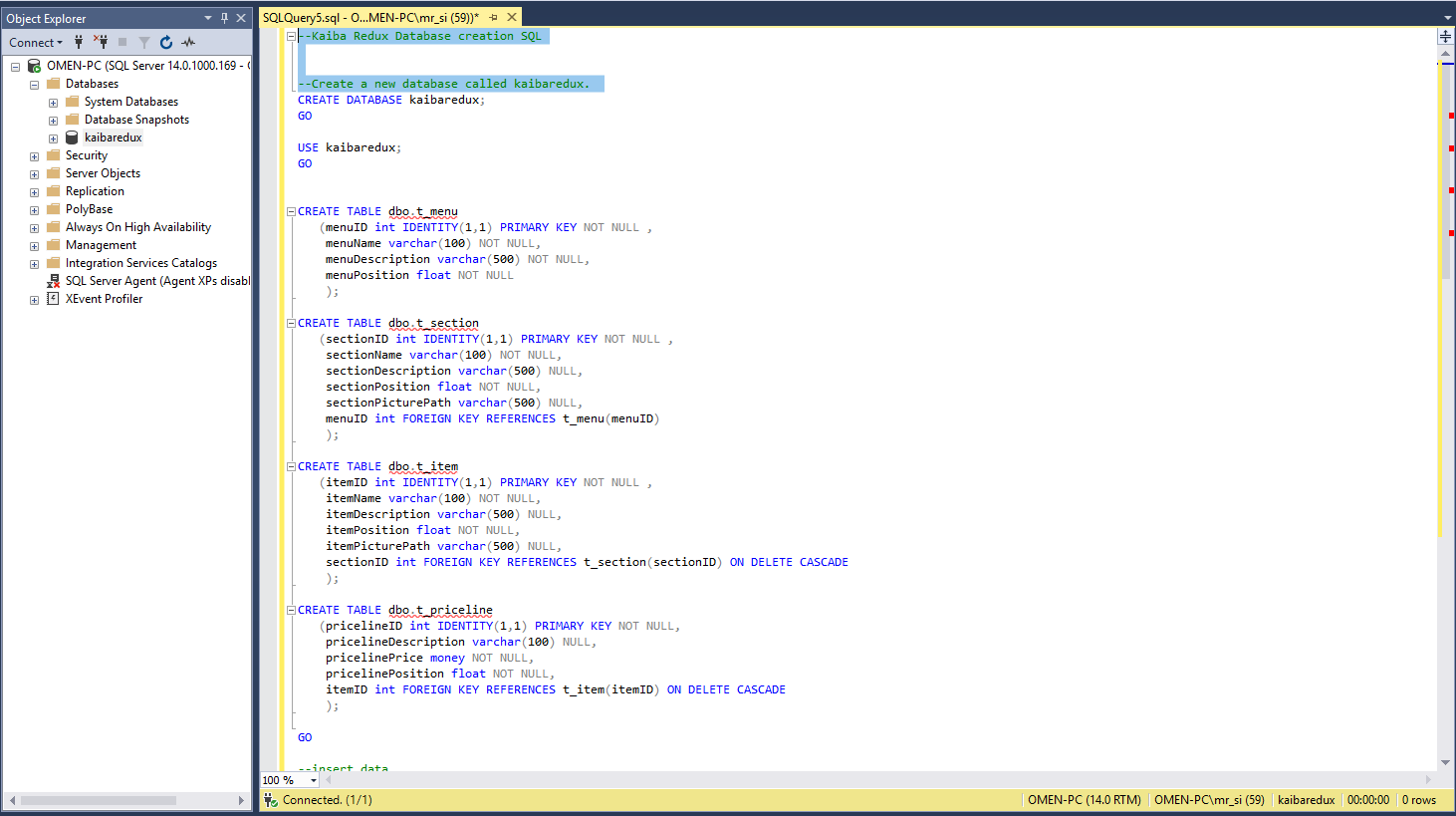
1. Open the **“KaibaReduxAPI”** folder and open **“Test Information reset SQL.txt”** and **select all** and **copy**



1. Open SQL Sever right click the **“Databases”** click **“New Database…”**
2. Name the database “kaibaredux”

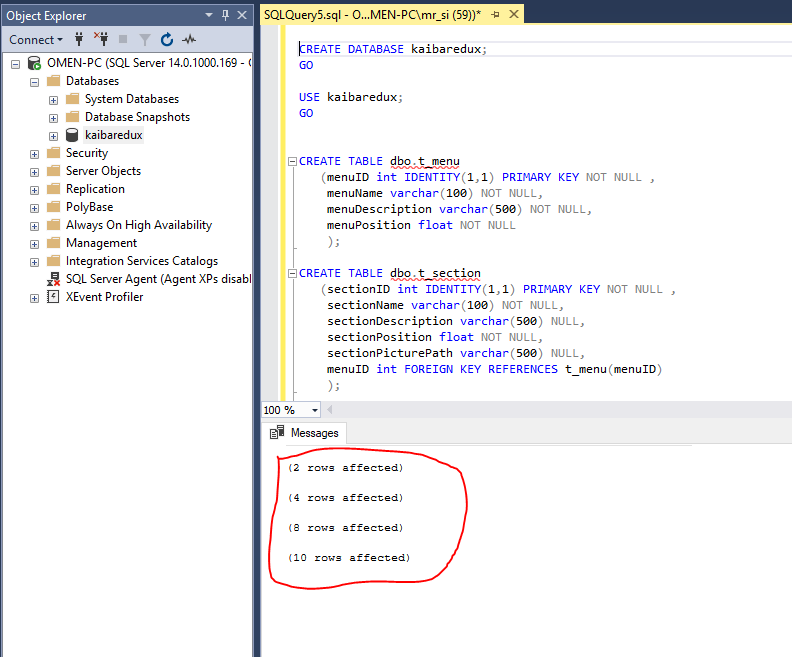


1. Right click the created **“kaibaredux”** database and click **“New Query”**
2. Paste the SQL commands from the text into the query
3. **click** **“Execute”** above

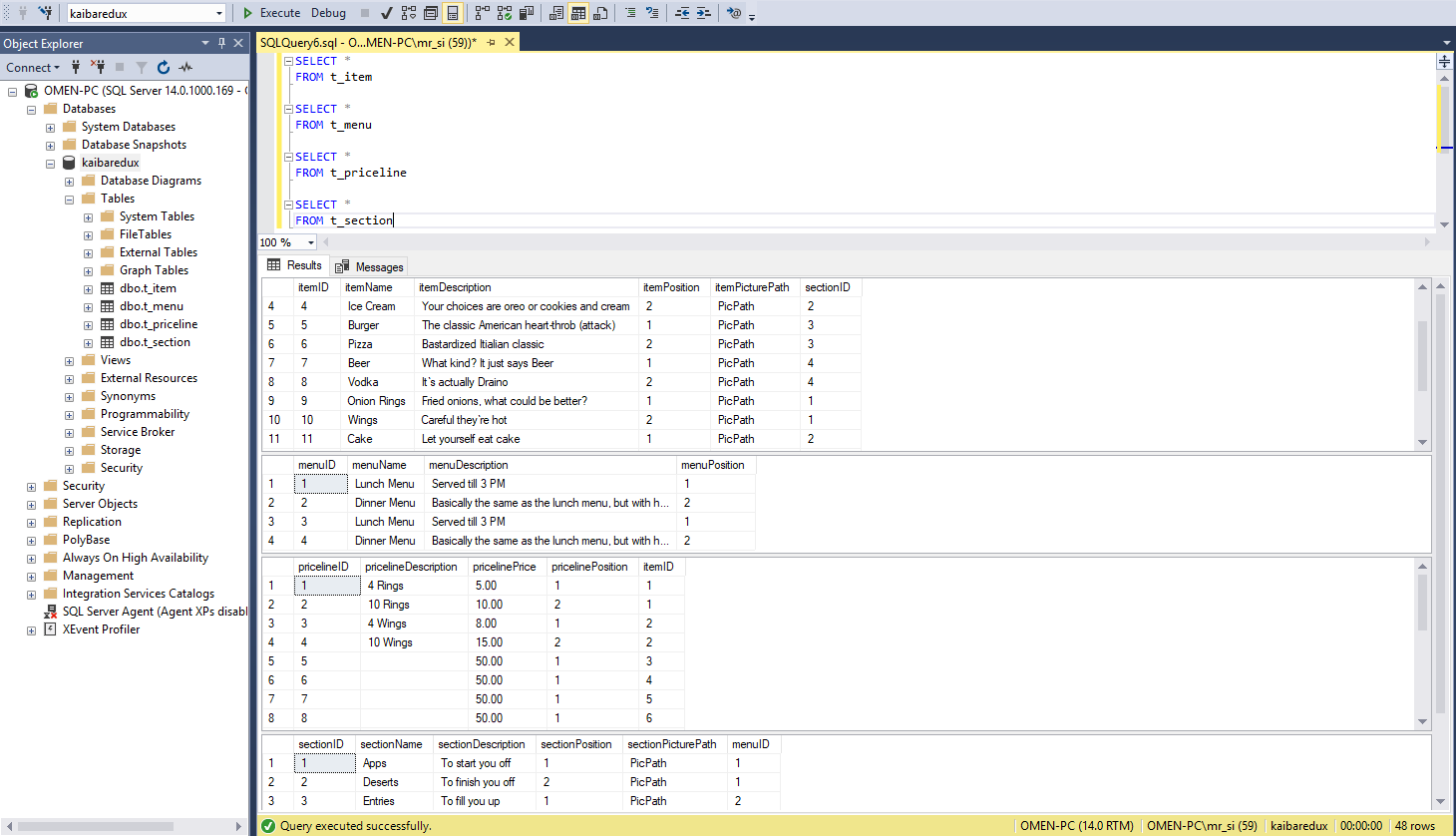


1. The database is properly created if the following is displayed with no errors below.

Do not panic if you have errors, most likely you have repeated this step in SQL Server. Redundant data entry is not allowed.



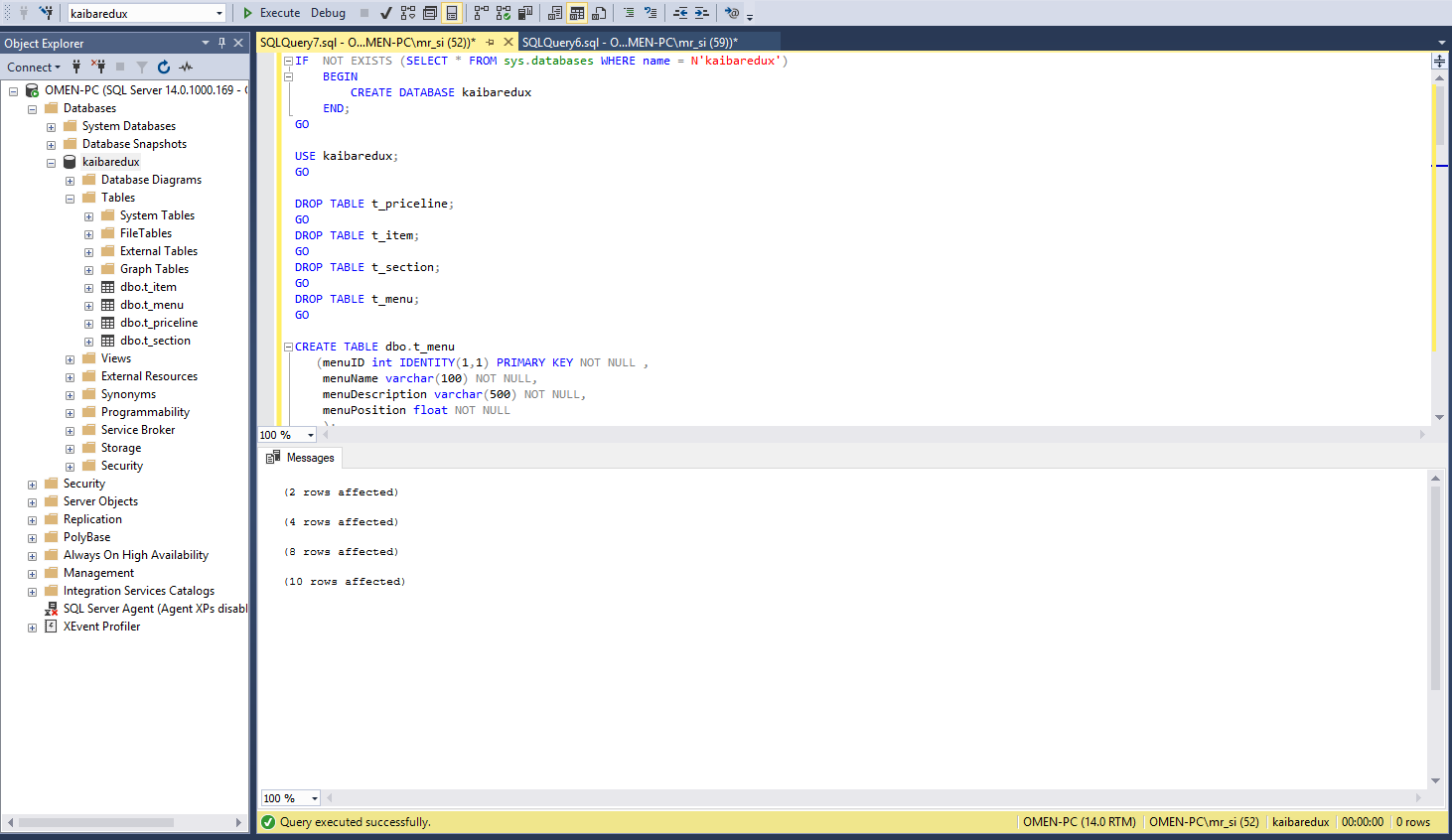
1. Simple SQL **SELECT** statements can display the table content



## **3.3 Resetting the KaibaRedux Database in SQL Server**

If for any reason data has been deleted from the SQL Server database, running the following reset SQL commands will restore the default data and remove any new data from the existing DB. This is particularly useful when testing functionality and afterwards you can easily reset the database to it’s original state.

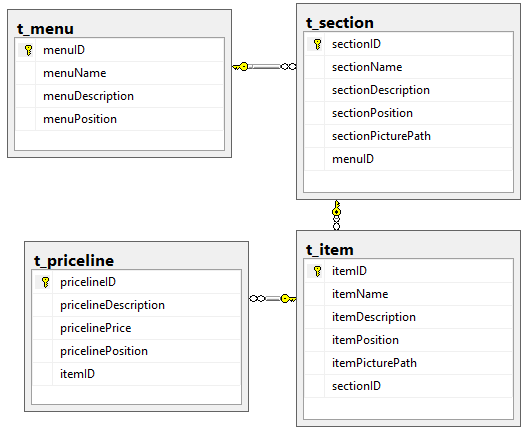
1. The SQL database reset commands can be found in **“KaibaReduxAPI”** folder in **“Test Information reset SQL.txt”**
2. Select **all** **the text** and repeat from **step 3** in **section 3.2**



# **4 Database Layer**

## **4.1 Database Structure**

The kaibaredux database has 4 tables: t\_menu, t\_section, t\_item, and t\_priceline. Their relationship is detailed in the following Entity Relationship Diagram (ERD)



The first attribute of each table is its primary key, which uniquely identifies each record in the table and are integers. Name and description hold the menu information and are strings. Position defines which the order in which records appear in the menu and is a double. PicturePath is a currently unused attribute that would have been used to store the path to each item’s image on the menu. t\_section, t\_item, and t\_priceline have another ID, which is the foreign key of its parent table. This allows us to associate records between tables.

Each priceline represents a different price option for an item. For example in the test data 4 onion rings are $5 and 10 rings are $10.

**NOTE**: The primary keys are all set to autoincrement, which means that IDs are all assigned automatically by the database. This means that the IDs **cannot** be specified when inserting new records

**4.2 Testing the Database**

--Try running some select statements on the database to make sure it’s working.

Ex. SELECT \* FROM t\_item

**NOTE**: It’s good practice to always use capital letters for SQL statements

--Try inserting a few records into the tables.

Ex. INSERT INTO t\_item

(itemName, itemDescription, itemPosition, itemPicturePath, sectionID) VALUES

('Sampler, 'A little bit of everything', 3, 'PicPath', 1)

For more info and examples of INSERT statements see: <https://www.w3schools.com/sql/sql_insert.asp>

--Try updating a few records.

Ex. UPDATE t\_item SET

itemName = 'Panko Breaded Onion Rings',

itemDescription = 'Updated to have better breading',

itemPosition = 1,

itemPicturePath = 'Updated Picture Path',

sectionID = 1

WHERE itemID = 1;

For more info and examples of UPDATE statements see: <https://www.w3schools.com/sql/sql_update.asp>

--Try deleting a few records

Ex. DELETE FROM t\_section

WHERE sectionID = 1

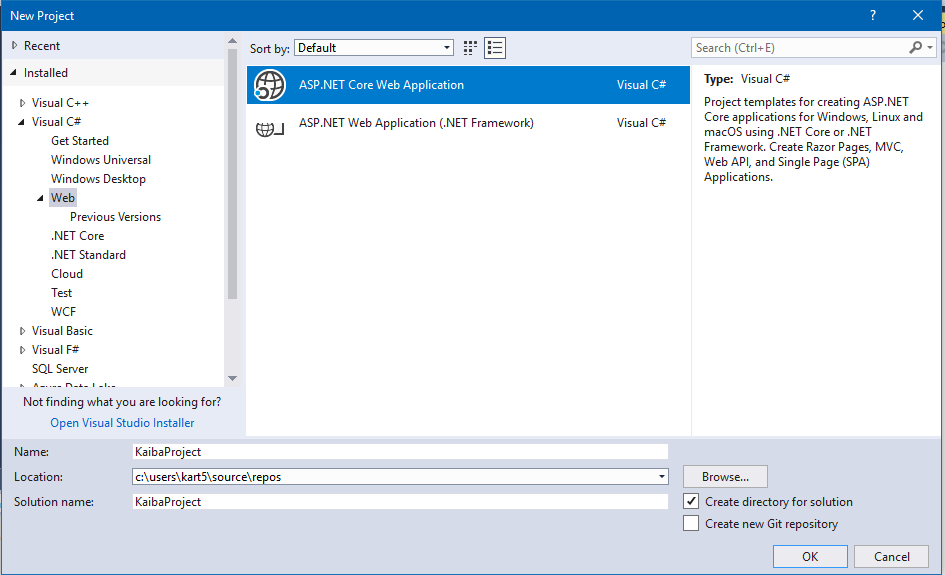
For more info and examples of DELETE statements see: <https://www.w3schools.com/sql/sql_delete.asp>

**REMEMBER**: You can always reset the database by running the statements found within Test Data Reset SQL.txt. (See Section 3.3)

## **5.1 Create New Project**

Create a new project in Visual Studio: File 🡪 New Project

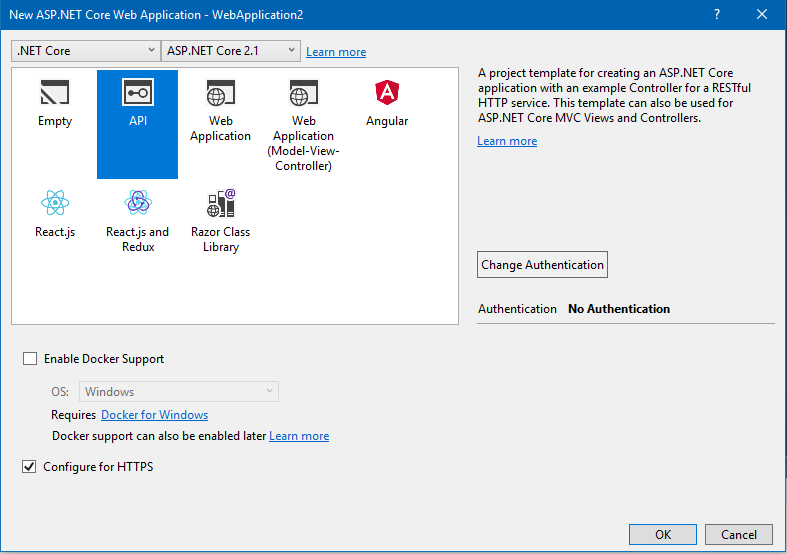
Select ASP.NET Core Web Application template. Name the project something nice. Click OK.



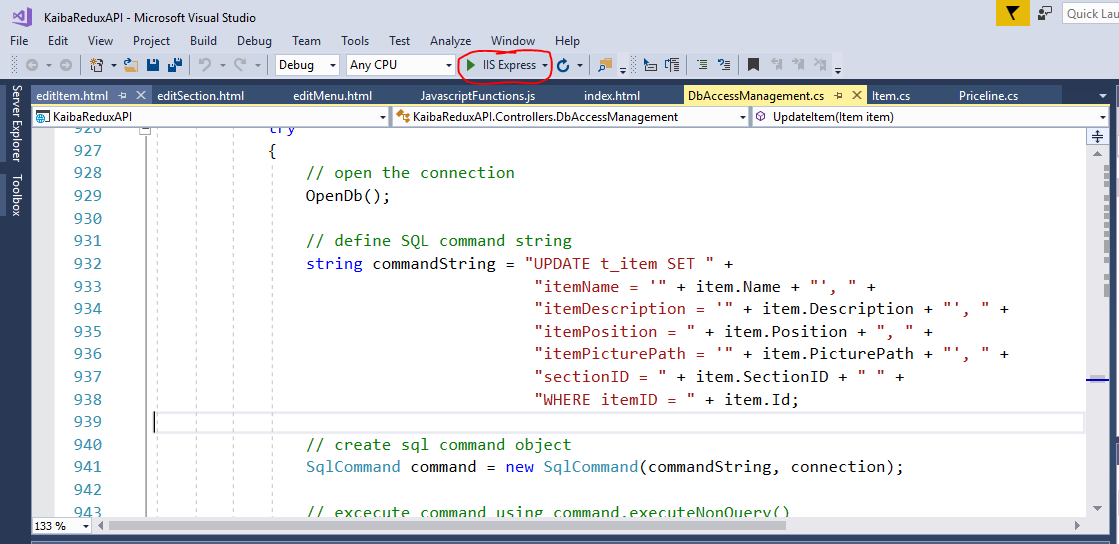
Select .NET Core. Select ASP.NET Core 2.1. Select API. Leave rest as default.

If you do not see ASP.NET Core 2.1, you need to install it here:

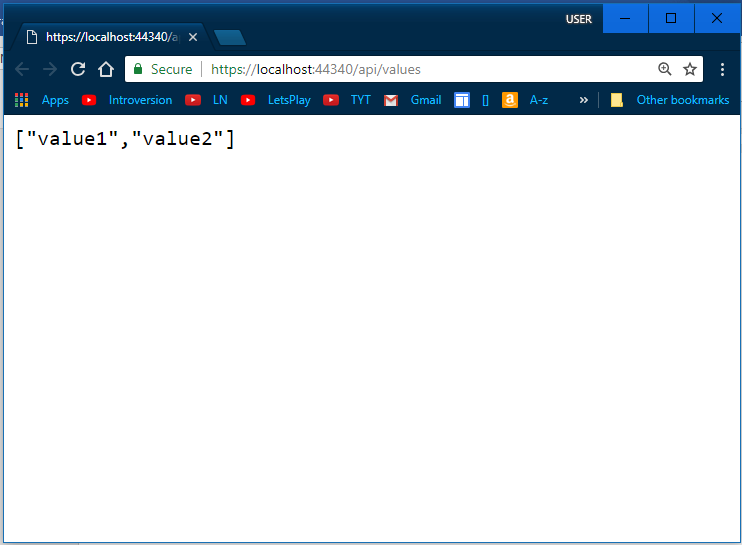
<https://www.microsoft.com/net/download/thank-you/dotnet-sdk-2.1.302-windows-x64-installer>



After Visual Studio finishes creating the project, test it by running it. On the tool bar press the green play triangle that says IIS express.



It should open your default web browser, which should go to the URL: https://localhost:44340/api/values, and display this text: ["value1","value2"]



Stop the project by clicking the red stop square (same location as previous button)

**NOTE**: If it opens in Internet Explorer, it will prompt you to download a file. If that’s the case use a different browser, why the heck are you even using IE?

## **5.2 To Be or Not To Be (Lazy)**

Programming is a practical skill. You don’t learn by staring at code or reading some developers guide. You learn by doing. If you really want to learn something, then you should create this project from scratch with a minimum of copying and pasting from this document or from the working version of this project. It’s going to be a good amount of work, but that’s how you learn by doing. That’s what we strongly recommend.

But let’s be honest, you’re probably going to be lazy and just copy and paste everything without trying to understand how it works. If you’re going to do that, at least read the comments for each function, so you get a feeling for how they work. The code is pretty well commented, so it shouldn’t be too hard to follow.

In the end, you get as much out of this project as you put in. You spend little to no time with it or have another group member do everything and you’ll find this to be a waste of time project where you learned very little. But if you put some effort into the project and put in the time to understand how its various pieces work, then you’ll learn some valuable coding skills.

## **6.1 Model Classes**

The first step is to create objects to represent each element of the menu. In the solution explorer on the right, create a new folder called Models.

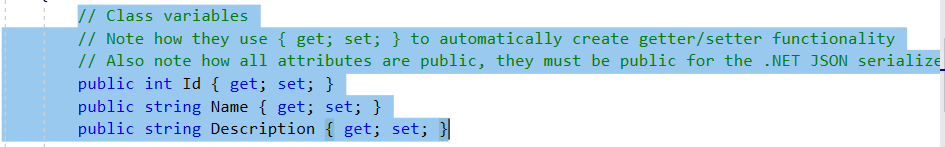
Right Click your project’s name 🡪 add 🡪 new Folder, name it Models

Copy the model classes from the working KaibaReduxAPI project into the newly created model folder. Each of these model classes contains the same attributes as the database tables do. In addition, menu has a list of sections, section has a list of items, and item has a list of pricelines.

Note how ASP.NET simplifies the definition of the variable and the creation of getters and setters. These lines create private variables that are accessed and set through public getters and setters. The getters and setters are accessed the same way as a normal variable

Ex. int x = menu.Id;

menu.Name = “Fred”:

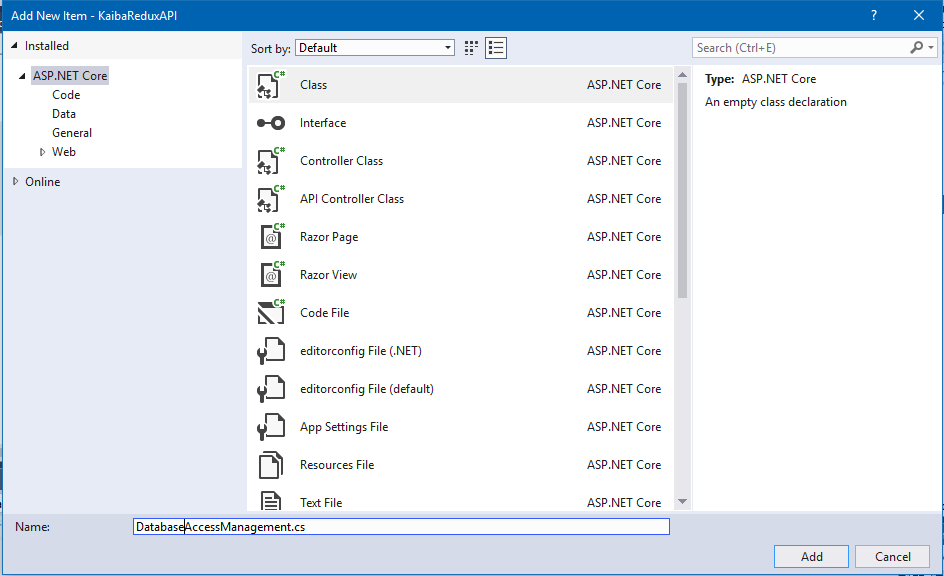


These model classes will be used to pass data back and forth between the API controllers and the Database Access Management class.

## **7.1 Database Access Management class**

Create a new controller class. Right click on the controllers folder 🡪 Add item

Select Class. Name it DatabaseAccessManagement.cs (the .cs signifies that it’s C# code)



At the top add these using (import) statements

*using System.Data;*

*using System.Data.SqlClient;*

*using NewProjectName.Models;*

*// where NewProjectName should be replaced by the name of your new project*

Add these lines inside the innermost brackets (after public class DatabaseAccessManagement):

*// This is the connection string that points to the database.*

*private static readonly string CONNECTION\_STRING = "Data Source=DESKTOP-PPEIFCP;Initial Catalog=kaibaredux;Trusted\_Connection=yes;";*

*// This is the SQL connection object, which is used to execute operations on the DB*

*private SqlConnection connection;*

As specified in Section 3.1, change DESKTOP-PPEIFCP in that line to the name of your SQL server instance, which you can find in SQL management studio.

## **7.2 Database Test Methods**

Now we need to test that the database connection works.

Create the DbTest() class:

*public static bool DBTest()*

*//a function to test whether the connection can be opened and closed without an error*

*{*

*// Declare a connection object*

*SqlConnection cnn;*

*// Instantiate connection object*

*// Give it the connection string constant*

*cnn = new SqlConnection(CONNECTION\_STRING);*

*// Try opening and closing the connection*

*try*

*{*

*cnn.Open();*

*cnn.Close();*

*// If connection opened and closed without errors, output a confirmation to the console*

*System.Diagnostics.Debug.WriteLine("DB-DEBUG: Connection worked");*

*return true;*

*}*

*catch (Exception ex)*

*{*

*// If there was an error, output it to the console*

*System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);*

*}*

*return false;*

*}*

In the Solution Explorer open ValuesController.cs

Modify the get() function, replace:

*return new string[] { "value1", "value2" };*

With:

if (DbAccessManagement.DBTest())

{

return new string[] { "connected" };

}

else

{

return new string[] { "failed" };

}

Run the project, it should display connected if everything is set up correctly. If it returns failed check that the connection string has been configured properly.

## **Note**: Whenever you have an issue remember to check the Output window, which appears while the project is running. Under ‘Show output from’ Select ‘Debug’. When it’s a database issue search for (ctrl-F) DB-DEBUG, which is printed before all errors in this class

## 

In DAM class, Create the DbTest2() method. This method retrieves all associated items and pricelines from the database and returns them as a string.

*public static string DBTest2()*

*// Try outputting information from the db to the page*

*{*

*SqlConnection cnn;*

*cnn = new SqlConnection(CONNECTION\_STRING);*

*try*

*{*

*string result = " ";*

*cnn.Open();*

*string sqlString = "SELECT \* FROM t\_item i, t\_priceline p WHERE i.itemID = p.itemID";*

*SqlCommand myCommand = new SqlCommand(sqlString, cnn);*

*SqlDataReader myReader = myCommand.ExecuteReader();*

*bool firstTime = true;*

*while (myReader.Read())*

*{*

*if (!firstTime)*

*{*

*result += ", ";*

*}*

*firstTime = false;*

*result += "[";*

*result += (myReader["itemName"].ToString()) + ",";*

*result += (myReader["itemDescription"].ToString()) + ",";*

*result += (myReader["itemPicturePath"].ToString()) + ",";*

*result += (myReader["pricelinePrice"].ToString()) + "";*

*result += "]";*

*}*

*result += " ";*

*cnn.Close();*

*System.Diagnostics.Debug.WriteLine("Connection worked");*

*return result;*

*}*

*catch (Exception ex)*

*{*

*System.Diagnostics.Debug.WriteLine(ex.Message);*

*return ex.Message;*

*}*

*}*

## Modify the get() method in the ValuesController again

{

return new string[] { DbAccessManagement.DBTest2() };

## }

Run the Project. It should display a list of items and prices from the database. If it does, congratulations you’ve got your API pulling data from the database.

## **7.3 Database Open and Close Methods**

Create the OpenDb() and closeDb() methods. These will be called before and after every database operation. OpenDb() creates the connection object that all other methods will use. CloseDb() closes the connection to free up resources. This is very important as only one connection can be open at a time.

private void OpenDb()

// Opens the database connection. This must be done before every db operation.

{

if (connection == null)

{

connection = new SqlConnection(CONNECTION\_STRING);

connection.Open();

}

}

private void CloseDb()

// Closes the database connection. This should be done after every database operation, whether it suceeded or not

// This is a general good programming practice, as it frees up system resources (makes sure you're not opening a new connection every time, but not closing them)

{

// check if the connection is already null

// If the connection was null and we tried to close it, we would get a NullPointerException

if (connection != null)

{

// if it isn't null, then we need to close it

connection.Close();

// and set it to null

connection = null;

}

}

## **7.4 The GetMenus() method**

Create the GetMenus() method. It is a well-commented example class that shows the basic structure of all the database operation methods. Particularly, note how it uses a DataReader to access the information retrieved from the database.

public List<Menu> getMenus()

// returns a list containing menu objects, which contain the information about each menu

{

// Declare a string list to hold the data we get from the DB

// Note how you must declare the list's data type: <string>

List<Menu> results = new List<Menu>();

// Use a try catch here because it's very likely that the connection could fail and throw an error

try

{

// open the connection

OpenDb();

// Define the SQL command statement

// Web simply want to retrieve all the menus ordered by the position field

string commandString = "SELECT \* FROM t\_menu " +

"ORDER BY menuPosition";

// Create the SQL command object, give it the command string and the connection object

SqlCommand command = new SqlCommand(commandString, connection);

// Execute the command, since this is a select use SqlCommand.ExecuteReader()

// It will return a SQLDataReader object, which we assign to the variable "dataReader"

SqlDataReader dataReader = command.ExecuteReader();

// A DataReader allows you to read one row at a time

// You can then call SqlDataReader.Read(), which will allow you to access the next row

// it returns true as long as there is another row to access

// it will return false when there are no further rows to access

// By placing the SqlDataReader.Read() call inside a while, we can keep reading the row data until there are no further rows

while (dataReader.Read())

{

// New Menu Object to store data

Menu menu = new Menu();

// Get each column from each row

// The ToString() method ensures that we recieve a string

menu.Id = (int) dataReader["menuID"];

menu.Name = dataReader["menuName"].ToString();

menu.Description = dataReader["menuDescription"].ToString();

menu.Position = (double) dataReader["menuPosition"];

// add that menu to the list

results.Add(menu);

}

}

catch (Exception ex)

// If there is an Exception (aka an error) then the catch block is executed

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

// If there was an error we still need to return something

// return an empty list

results = new List<Menu>();

}

finally

{

// whether there was an error or not, we need to close the connection

// that's what the finally block is for

CloseDb();

}

// lastly return the result

// it's good practice to always have only a single return statement at the end of the method

return results;

}

**7.5 The getObject() Methods**

The GetMenu(), GetSection(), GetItem(), and GetPriceline() methods allow us to select a specific record by it’s unique Id. Notice that GetMenu() calls GetSectionsInMenu(id) in order to populate the menu ojbect’s list of sections. Similarly GetSection() calls GetItemsInSection(id) and GetItem() calls getPricelinesForItem(id). Thus the objects returned by these methods contain all their associated sections, items, and or pricelines.

public Menu getMenu(int id)

// takes a menu id and returns a corresponding menu object that contains it's sections, which contain items, which contain pricelines

// if that menu is not found, returns null

{

// Declare return variable

Menu result = new Menu();

// Use a try catch here because it's very likely that the connection could fail and throw an error

try

{

// open the connection

OpenDb();

// Define the SQL command statement

// Web simply want to retrieve a specific menu

string commandString = "SELECT \* FROM t\_menu " +

"WHERE menuID = " + id;

// Create the SQL command object, give it the command string and the connection object

SqlCommand command = new SqlCommand(commandString, connection);

// Execute the command, since this is a select use SqlCommand.ExecuteReader()

// It will return a SQLDataReader object, which we assign to the variable "dataReader"

SqlDataReader dataReader = command.ExecuteReader();

// Since we are retriving a single row, we can use an if statement

if (dataReader.Read())

{

// There was a row returned, so we can get each column data from each row

result.Id = (int)dataReader["menuID"];

result.Name = dataReader["menuName"].ToString();

result.Description = dataReader["menuDescription"].ToString();

result.Position = (double)dataReader["menuPosition"];

// Close the DataReader

dataReader.Close();

// Now we need to get the sections in this menu

result.SectionList = GetSectionsInMenu(result.Id);

}

else

{

// no row was returned, so the menu was not found

// in that case we return null, to signify that nothing was found

result = null;

}

}

catch (Exception ex)

// If there is an Exception (aka an error) then the catch block is executed

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

// If there was an error we still need to return something

// return an empty menu, with the name Database ERROR

result = new Menu();

result.Name = "Database ERROR";

}

finally

{

// whether there was an error or not, we need to close the connection

// that's what the finally block is for

CloseDb();

}

// lastly return the result

// it's good practice to always have only a single return statement at the end of the method

return result;

}

public Section GetSection(int id)

// takes a section ID and returns a section object

// returns null if not found

{

// the list to hold results from the database and eventually return

Section result = new Section();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_section " +

"WHERE sectionID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

if (dataReader.Read())

{

// new section object to hold data

Section sect = new Section();

// get data from the dataReader

sect.Id = (int)dataReader["sectionID"];

sect.Name = dataReader["sectionName"].ToString();

sect.Description = dataReader["sectionDescription"].ToString();

sect.Position = (double)dataReader["sectionPosition"];

sect.PicturePath = dataReader["sectionPicturePath"].ToString();

sect.MenuID = (int)dataReader["menuID"];

// close the DataReader

dataReader.Close();

// get this section's items

sect.ItemList = GetItemsInSection(sect.Id);

// assign section object to the result to be returned

result = sect;

}

else

{

// no row was returned, so the section was not found

// in that case we return null, to signify that nothing was found

result = null;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// return a section with the name Database ERROR

result = new Section();

result.Name = "Database ERROR";

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return result;

}

public Item GetItem(int id)

// takes an item ID and returns an item object

// returns null if not found

{

// the list to hold results from the database and eventually return

Item result = new Item();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_item " +

"WHERE itemID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

if (dataReader.Read())

{

// new section object to hold data

Item item = new Item();

// get data from the dataReader

item.Id = (int)dataReader["itemID"];

item.Name = dataReader["itemName"].ToString();

item.Description = dataReader["itemDescription"].ToString();

item.Position = (double)dataReader["itemPosition"];

item.PicturePath = dataReader["itemPicturePath"].ToString();

item.SectionID = (int)dataReader["sectionID"];

// close the DataReader

dataReader.Close();

// get this item's pricelines

item.PriceLineList = getPricelinesForItem(item.Id);

// assign item object to the result to be returned

result = item;

}

else

{

// no row was returned, so the item was not found

// in that case we return null, to signify that nothing was found

result = null;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// return an item with the name Database ERROR

result = new Item();

result.Name = "Database ERROR";

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return result;

}

public Priceline GetPriceline(int id)

// takes a pricelineID and returns a Priceline object

// returns null if not found

{

// the list to hold results from the database and eventually return

Priceline result = new Priceline();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_priceline " +

"WHERE pricelineID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

if (dataReader.Read())

{

// new section object to hold data

Priceline price = new Priceline();

// get data from the dataReader

price.Id = (int)dataReader["pricelineID"];

price.Description = dataReader["pricelineDescription"].ToString();

price.Price = (decimal)dataReader["pricelinePrice"];

price.Position = (double)dataReader["pricelinePosition"];

price.ItemID = (int)dataReader["itemID"];

// close the DataReader

dataReader.Close();

// assign result

result = price;

}

else

{

// wasn't found, return null

result = null;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// return a priceline with the description Database ERROR

result = new Priceline();

result.Description = "Database ERROR";

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return result;

}

## **7.6 The GetObjectInObject() methods**

GetSectionsInMenu() takes a menu ID and returns a list of sections in that menu. Those sections will in turn contain their respective items. So essentially it returns everything on that menu. It calls GetItemsInSection() which returns that list of items. Which in turn calls GetPricelinesForItem(), which returns a list of priceline objects.

**Note**: Because each of these functions is calling another function that will use the same global connection object, you must close the DataReader object before calling the next function. Also note that because of these nested function calls OpenDb() maybe called when the connection is still open. This is fine as OpenDb() only opens the connection if it’s already closed. Similarly CloseDb() only closes the connection if it’s currently open.

public List<Section> GetSectionsInMenu(int id)

// takes a menuID and returns a list containing all the sections in that menu

// each section will in turn contain it's corresponding items

// in order to get this list of it's items, it calls getItemsInSection()

{

// the list to hold results from the database and eventually return

List<Section> results = new List<Section>();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_section " +

"WHERE menuID = " + id + " " +

"ORDER BY sectionPosition";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

while (dataReader.Read())

{

// new section object to hold data

Section sect = new Section();

// get data from the dataReader

sect.Id = (int) dataReader["sectionID"];

sect.Name = dataReader["sectionName"].ToString();

sect.Description = dataReader["sectionDescription"].ToString();

sect.Position = (double) dataReader["sectionPosition"];

sect.PicturePath = dataReader["sectionPicturePath"].ToString();

sect.MenuID = (int)dataReader["menuID"];

// put object in list

results.Add(sect);

}

// close the DataReader

dataReader.Close();

// use a foreach loop, to call getItemsInSection for each object

foreach (Section s in results)

{

s.ItemList = GetItemsInSection(s.Id);

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set results to be an empty list

results = new List<Section>();

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return results;

}

private List<Item> GetItemsInSection(int id)

// takes a sectionID and returns a list of all the items in that section

// each item will contain it's own price lines

{

// the list to hold results from the database and eventually return

List<Item> results = new List<Item>();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_item " +

"WHERE sectionID = " + id + " " +

"ORDER BY itemPosition";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

while (dataReader.Read())

{

// new Item object to hold data

Item item = new Item();

// get data from dataReader

item.Id = (int)dataReader["itemID"];

item.Name = dataReader["itemName"].ToString();

item.Description = dataReader["itemDescription"].ToString();

item.Position = (double)dataReader["itemPosition"];

item.PicturePath = dataReader["itemPicturePath"].ToString();

item.SectionID = (int)dataReader["sectionID"];

// put Item in list

results.Add(item);

}

// close the DataReader

dataReader.Close();

// use a foreach to call getPricelinesForItem() on each item

foreach (Item i in results)

{

i.PriceLineList = getPricelinesForItem(i.Id);

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// assign empty list to results

results = new List<Item>();

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return results;

}

private List<Priceline> getPricelinesForItem(int id)

// takes an item ID and returns a list of all pricelines associated with that item

{

// the list to hold results from the database and eventually return

List<Priceline> results = new List<Priceline>();

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "SELECT \* FROM t\_priceLine " +

"WHERE itemID = " + id + " " +

"ORDER BY pricelinePosition";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command and assign results to a dataReader

SqlDataReader dataReader = command.ExecuteReader();

// while loop to get all row data

while (dataReader.Read())

{

// new Priceline object to hold data

Priceline price = new Priceline();

// get data from dataReader

price.Id = (int)dataReader["pricelineID"];

price.Description = dataReader["pricelineDescription"].ToString();

price.Price = (decimal)dataReader["pricelinePrice"];

price.Position = (double)dataReader["pricelinePosition"];

price.ItemID = (int)dataReader["itemID"];

// put Priceline in list

results.Add(price);

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// assign empty list to results

results = new List<Priceline>();

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the results

return results;

}

## **7.8 The INSERT and UPDATE Statements**

These are pretty simple. Each INSERTs differs only in the SQL statement and the respective input it takes. Note that for INSERTs the given primary key id does not matter, because the database will assign a unique id. UPDATEs are very similar to INSERTs, differing only in the SQL statements and that here the primary key id does matter, because it is how we identify the record that is being updated.

**Note**: That since these SQL queries are operations that will change the database they will not return a dataReader (as there is no data to return), but instead will return a number of rows affected. Also note that they used command.ExecuteNonQuery(), instead of command.ExecuteReader();

public bool InsertMenu (Menu menu)

// takes a menu object and creates a coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// Note how this uses dynamic SQL and drops values directly into the command string

// However, this leaves the application vulnerable to SQL injection

// using prepared statements (aka parameterized) would be a better solution

string commandString = "INSERT INTO t\_menu (menuName, menuDescription, menuPosition) " +

"VALUES ('" + menu.Name + "', '" + menu.Description + "', " + menu.Position + ")";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool InsertSection(Section section)

// takes a Section object and creates a coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// Note how this uses dynamic SQL and drops values directly into the command string

// However, this leaves the application vulnerable to SQL injection

// using prepared statements (aka parameterized) would be a better solution

string commandString = "INSERT INTO t\_section (sectionName, sectionDescription, sectionPosition, sectionPicturePath, menuID) " +

"VALUES ('" + section.Name + "', '" + section.Description + "', " + section.Position + ", '" + section.PicturePath + "', " + section.MenuID + ")";

System.Diagnostics.Debug.WriteLine("AAAAAAAAAAAAAAAA: " + commandString);

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool InsertItem(Item item)

// takes a Item object and creates a coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// Note how this uses dynamic SQL and drops values directly into the command string

// However, this leaves the application vulnerable to SQL injection

// using prepared statements (aka parameterized) would be a better solution

string commandString = "INSERT INTO t\_item (itemName, itemDescription, itemPosition, itemPicturePath, sectionID) " +

"VALUES ('" + item.Name + "', '" + item.Description + "', " + item.Position + ", '" + item.PicturePath + "', " + item.SectionID + ")";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool InsertPriceline(Priceline price)

// takes a Priceline object and creates a coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// Note how this uses dynamic SQL and drops values directly into the command string

// However, this leaves the application vulnerable to SQL injection

// using prepared statements (aka parameterized) would be a better solution

string commandString = "INSERT INTO t\_priceline (pricelineDescription, pricelinePrice, pricelinePosition, itemID) " +

"VALUES ('" + price.Description + "', " + price.Price + ", " + price.Position + ", " + price.ItemID + ")";

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool UpdateMenu(Menu menu)

// takes a menu object and updates the coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "UPDATE t\_menu SET " +

"menuName = '" + menu.Name + "', " +

"menuDescription = '" + menu.Description + "', " +

"menuPosition = " + menu.Position + " " +

"WHERE menuID = " + menu.Id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery()

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool UpdateSection(Section section)

// takes a Section object and updates the coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "UPDATE t\_section SET " +

"sectionName = '" + section.Name + "', " +

"sectionDescription = '" + section.Description + "', " +

"sectionPosition = " + section.Position + ", " +

"sectionPicturePath = '" + section.PicturePath + "', " +

"menuID = " + section.MenuID + " " +

"WHERE sectionID = " + section.Id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery()

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool UpdateItem(Item item)

// takes an Item object and updates the coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "UPDATE t\_item SET " +

"itemName = '" + item.Name + "', " +

"itemDescription = '" + item.Description + "', " +

"itemPosition = " + item.Position + ", " +

"itemPicturePath = '" + item.PicturePath + "', " +

"sectionID = " + item.SectionID + " " +

"WHERE itemID = " + item.Id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery()

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool UpdatePriceline(Priceline price)

// takes an Priceline object and updates the coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "UPDATE t\_priceline SET " +

"pricelineDescription = '" + price.Description + "', " +

"pricelinePrice = " + price.Price + ", " +

"pricelinePosition = " + price.Position + ", " +

"itemID = " + price.ItemID + " " +

"WHERE pricelineID = " + price.Id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery()

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

## **7.9 The DELETE Statements**

These are very simple. Each DELETE simply takes a primary key id, which it then uses to execute a statement to delete the corresponding record. Note that if you delete a section any items and pricelines associated with it will also be deleted (a cascade delete). Similarly deleting an item will also delete it’s pricelines. But deleting a menu is not possible without first removing all it’s sections.

public bool DeleteMenu(int id)

// takes a menu id and deletes the coresponding database entry for it

// will not work if the menu still has sections in it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// if this menu has any sections in it, this will fail

string commandString = "DELETE FROM t\_menu " +

"WHERE menuID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool DeleteSection(int id)

// takes a section id and deletes the coresponding database entry for it

// will not work if the section still has items in it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// if this section has any items in it, this will fail

string commandString = "DELETE FROM t\_section " +

"WHERE sectionID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool DeleteItem(int id)

// takes an Item id and deletes the coresponding database entry for it

// will not work if the item still has pricelines assigned to it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

// if this item has any pricelines in it, this will fail

string commandString = "DELETE FROM t\_item " +

"WHERE itemID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

public bool DeletePriceline(int id)

// takes a priceline id and deletes the coresponding database entry for it

{

// this boolean will represent whether the operation was successful or not

bool result = false;

// try block to contain DB access statements

try

{

// open the connection

OpenDb();

// define SQL command string

string commandString = "DELETE FROM t\_priceline " +

"WHERE pricelineID = " + id;

// create sql command object

SqlCommand command = new SqlCommand(commandString, connection);

// excecute command using command.executeNonQuery() because this will not return a DataReader

// command.executeNonQuery() returns the number of rows affected, so assign that to a variable

int rowsAffected = command.ExecuteNonQuery();

// if one row was affected, then we were successful

if (rowsAffected == 1)

{

result = true;

}

}

// catch block to handle any errors

catch (Exception ex)

{

// Write the error to the console

// The "DB-DEBUG:" is just there to make finding that message in the console easier

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.Message);

System.Diagnostics.Debug.WriteLine("DB-DEBUG: " + ex.StackTrace);

// set result to false

result = false;

}

// finally block in which we close the connection, whether or not there was an error

finally

{

CloseDb();

}

// lastly return the result

return result;

}

## **7.10 SQL Injection Vulnerability**

All of the queries used in this project use dynamic SQL like the following

// define SQL command string

string commandString = "DELETE FROM t\_priceline " +

"WHERE pricelineID = " + id;

This is actually a bad practice, because these query strings are created by literally dropping data into the middle of the string. This works fine, but it leaves your application vulnerable to SQL injection. SQL injection can allow a malicious hacker to do anything they want to your database.

For more info see: <https://www.w3schools.com/sql/sql_injection.asp>

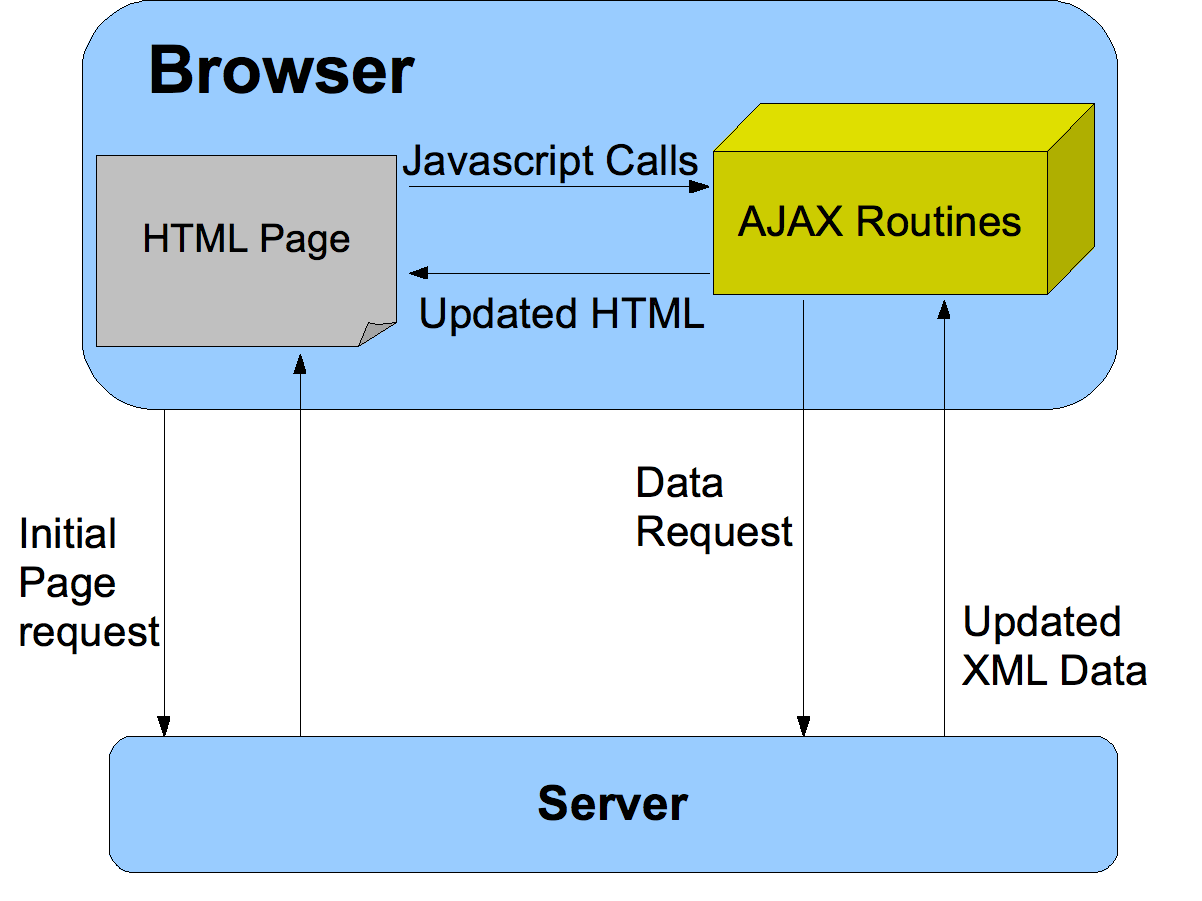
This application would be better secured using parameterized statements (aka prepared statements). These are not only generally faster, but also much more secure and make SQL injection almost impossible. For more info on prepared statements see:

<https://csharp-station.com/Tutorial/AdoDotNet/Lesson06>

## **8 API Controllers**

## **8.1 How the API works**

JSON (JavaScript Object Notation) is a well known and cross-platform compatible format for sending data over the internet. It allows us to take an object and easily express it as a string, which is much easier to pass over a network. One of the main benefits of JSON is that a web browser will automatically convert any JSON received over the network into a useable JavaScript object. It’s also very easy to create a JS object and then turn that object into a string of JSON. See diagram below:



**JSON Data**

ASP.NET’s (requires .NET Core 2.1) API functionalities provide a similar conversion between C# objects and JSON. Notice that in the values controller, get() returns ActionResult<IEnumerable<string>> .

An ActionResult is simply an HTTP Response of some kind. It could include returning a 404 Error Code or a web page to display. The <IEnumerable<string> part shows that it could also return a list of strings (or a string array). If a list of strings is returned ASP.NET will automatically convert that list of strings into JSON that is then sent as an HTTP response.

It goes further than that: a return type might be ActionResult<List<Menu>>. This means that we could return a list of menu objects, which ASP.NET will convert the list into JSON.

Similarly, if an HTTP request contains a message body of JSON, ASP.net will convert it into a specified object.

## **8.2 The MenuController**

Under the controllers folder create a new API controller class: MenuController.cs

This controller adds five new API functionalities. You can test 2 of them with your browser. See instructions at end of this section.

Replace auto-generated code with:

Make sure to replace YOUR\_PROJECT\_NAME with your project name

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using YOUR\_PROJECT\_NAME.Models;

// YOUR\_PROJECT\_NAME should be replaced by your project name

namespace KaibaReduxAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class MenuController : ControllerBase

{

// GET api/menu

[HttpGet]

public ActionResult<List<Menu>> GetMenus()

// returns a list of the menus

{

// create DbAccessManagement object

DbAccessManagement DAM = new DbAccessManagement();

// return the list of menus

System.Diagnostics.Debug.WriteLine("MenuCont: GetMenus() ");

return DAM.getMenus();

}

[HttpGet("{id}", Name = "GetMenu")] // Route = /api/menu/2

public ActionResult<Menu> GetMenu(int id)

// takes a menu id as a url parameter and returns a menu object with the corresponding information

// the menu will contain sections, which contain items, which contain pricelines

{

DbAccessManagement DAM = new DbAccessManagement();

// get the menu

Menu menu = DAM.getMenu(id);

// if it's null, then the menu wasn't found

if (menu == null)

{

// return a 404 ERROR

return NotFound();

}

else //otherwise return the menu

{

return menu;

}

}

[HttpPost] // Route = /api/menu

public IActionResult CreateMenu(Menu menu)

// POST request that takes JSON from the request body and builds a Menu object

// returns NoContent (201) if successful, returns server error (500) if unsuccessful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.InsertMenu(menu);

if (result)

{

return StatusCode(201);

}

else

{

return StatusCode(500);

}

}

[HttpDelete] // Route = DELETE /api/menu

// but uses the DELETE method (as opposed to the usual GET or POST

public IActionResult DeleteMenu(Menu menu)

// takes a menu object from the JSON body and deletes that record

// it only requires the id field, and ignores everything else

// returns NotFound (404) if unsuccessful, returns NoContent (201) if successful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.DeleteMenu(menu.Id);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

[HttpPut] // route = PUT /api/menu

// Note that it uses PUT instead of GET or POST

public IActionResult UpdateMenu(Menu menu)

// takes a menu object from the JSON body and updates that record

// returns NoContent (201) if successful, NotFound (404) if not

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.UpdateMenu(menu);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

}

}

After creating the MenuController and replacing the code, run the project.

Change the last part of the URL (after the port number) from api/values to api/menu. This should display information about all the menus.

Go to api/menu/1. That should display all information about the first menu, including it’s sections and items. (The URL pattern api/menu/#, will query for a menu with id = #)

To test the other functionality will need to use POSTMAN, see Section 9.

## **8.3 The SectionController**

Create a new API controller called SectionController.cs   
Replace code with:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using YOUR\_PROJECT\_NAME.Models;

// YOUR\_PROJECT\_NAME should be replaced by your project name

namespace KaibaReduxAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class SectionController : ControllerBase

{

[HttpGet("{id}", Name = "GetSection")] // Route = /api/section/2

public ActionResult<Section> GetSection(int id)

// takes a section id as a url parameter and returns a section object with the corresponding information

// the section will contain items, which contain pricelines

{

DbAccessManagement DAM = new DbAccessManagement();

// get the section

Section section = DAM.GetSection(id);

// if it's null, then the section wasn't found

if (section == null)

{

// return a 404 ERROR

return NotFound();

}

else //otherwise return the section

{

return section;

}

}

[HttpPost] // Route = /api/section

public IActionResult CreateSection(Section section)

// POST request that takes JSON from the request body and builds a Section object

// returns NoContent (201) if successful, returns server error (500) if unsuccessful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.InsertSection(section);

if (result)

{

return StatusCode(201);

}

else

{

return StatusCode(500);

}

}

[HttpDelete] // Route = DELETE /api/section

// but uses the DELETE method (as opposed to the usual GET or POST

public IActionResult DeleteSection(Section section)

// takes a Section object from the JSON body and deletes that record

// it only requires the id field, and ignores everything else

// returns NotFound (404) if unsuccessful, returns NoContent (201) if successful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.DeleteSection(section.Id);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

[HttpPut] // route = PUT /api/section

// Note that it uses PUT instead of GET or POST

public IActionResult UpdateSection(Section section)

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.UpdateSection(section);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

}

## } **8.4 The ItemController**

Create a new API controller called ItemController.cs

Replace code with:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using YOUR\_PROJECT\_NAME.Models;

// YOUR\_PROJECT\_NAME should be replaced by your project name

namespace KaibaReduxAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ItemController : ControllerBase

{

[HttpGet("{id}", Name = "GetItem")] // Route = /api/item/2

public ActionResult<Item> GetItem(int id)

// takes a Item id as a url parameter and returns a Item object with the corresponding information

// the Item will contain pricelines

{

DbAccessManagement DAM = new DbAccessManagement();

// get the Item

Item item = DAM.GetItem(id);

// if it's null, then the Item wasn't found

if (item == null)

{

// return a 404 ERROR

return NotFound();

}

else //otherwise return the Item

{

return item;

}

}

[HttpPost] // Route = /api/item

public IActionResult CreateItem(Item item)

// POST request that takes JSON from the request body and builds a Item object

// returns NoContent (201) if successful, returns server error (500) if unsuccessful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.InsertItem(item);

if (result)

{

return StatusCode(201);

}

else

{

return StatusCode(500);

}

}

[HttpDelete] // Route = DELETE /api/item

// but uses the DELETE method (as opposed to the usual GET or POST

public IActionResult DeleteItem(Item item)

// takes a Item object from the JSON body and deletes that record

// it only requires the id field, and ignores everything else

// returns NotFound (404) if unsuccessful, returns NoContent (201) if successful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.DeleteItem(item.Id);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

[HttpPut] // route = PUT /api/item

// Note that it uses PUT instead of GET or POST

public IActionResult UpdateItem(Item item)

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.UpdateItem(item);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

}

}

**8.5 The PricelineController**

Create a new API controller: PricelineController.cs

Replace code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using YOUR\_PROJECT\_NAME.Models;

// YOUR\_PROJECT\_NAME should be replaced by your project name

namespace KaibaReduxAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class PricelineController : ControllerBase

{

[HttpGet("{id}", Name = "GetPriceline")] // Route = /api/priceline/2

public ActionResult<Priceline> GetPriceline(int id)

// takes a Priceline id as a url parameter and returns a Priceline object with the corresponding information

// the Priceline will contain pricelines

{

DbAccessManagement DAM = new DbAccessManagement();

// get the Priceline

Priceline priceline = DAM.GetPriceline(id);

// if it's null, then the Priceline wasn't found

if (priceline == null)

{

// return a 404 ERROR

return NotFound();

}

else //otherwise return the Priceline

{

return priceline;

}

}

[HttpPost] // Route = /api/priceline

public IActionResult CreatePriceline(Priceline priceline)

// POST request that takes JSON from the request body and builds a Priceline object

// returns NoContent (201) if successful, returns server error (500) if unsuccessful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.InsertPriceline(priceline);

if (result)

{

return StatusCode(201);

}

else

{

return StatusCode(500);

}

}

[HttpDelete] // Route = DELETE /api/priceline

// but uses the DELETE method (as opposed to the usual GET or POST

public IActionResult DeletePriceline(Priceline priceline)

// takes a Priceline object from the JSON body and deletes that record

// it only requires the id field, and ignores everything else

// returns NotFound (404) if unsuccessful, returns NoContent (201) if successful

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.DeletePriceline(priceline.Id);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

[HttpPut] // route = PUT /api/priceline

// Note that it uses PUT instead of GET or POST

public IActionResult UpdatePriceline(Priceline priceline)

{

DbAccessManagement DAM = new DbAccessManagement();

bool result = DAM.UpdatePriceline(priceline);

if (result)

{

return NoContent();

}

else

{

return NotFound();

}

}

}

}

## **9 Testing the API with POSTMAN**

**9.1 Did you follow all the installation instructions?**

If not and you don’t have POSTMAN yet, download and install it here:

<https://www.getpostman.com/>

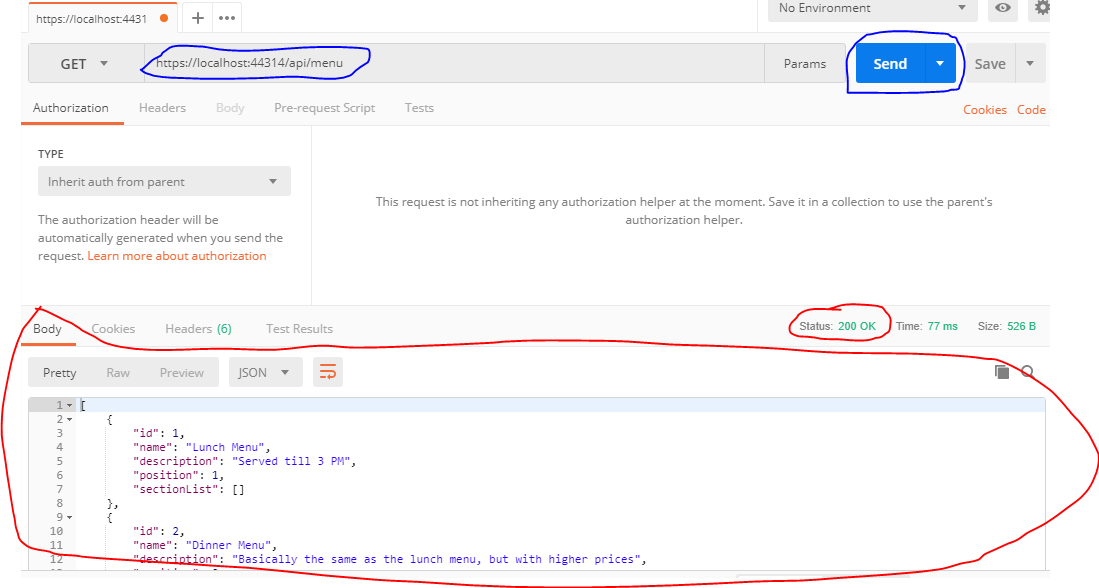
POSTMAN emulates HTML requests and responses, allowing you to test the API without a client.

**9.2 Try a couple manual POSTMAN requests**

**MAKE SURE** that the project is running or else none of this will work.

You should run the SQL reset statements before completely this part.

Click off of the welcome screen. In the top center enter the URL shown in your browser upon starting the project, should be like: *https://localhost:44314/api/menu* (might have a different port number). Then click send to send the request.

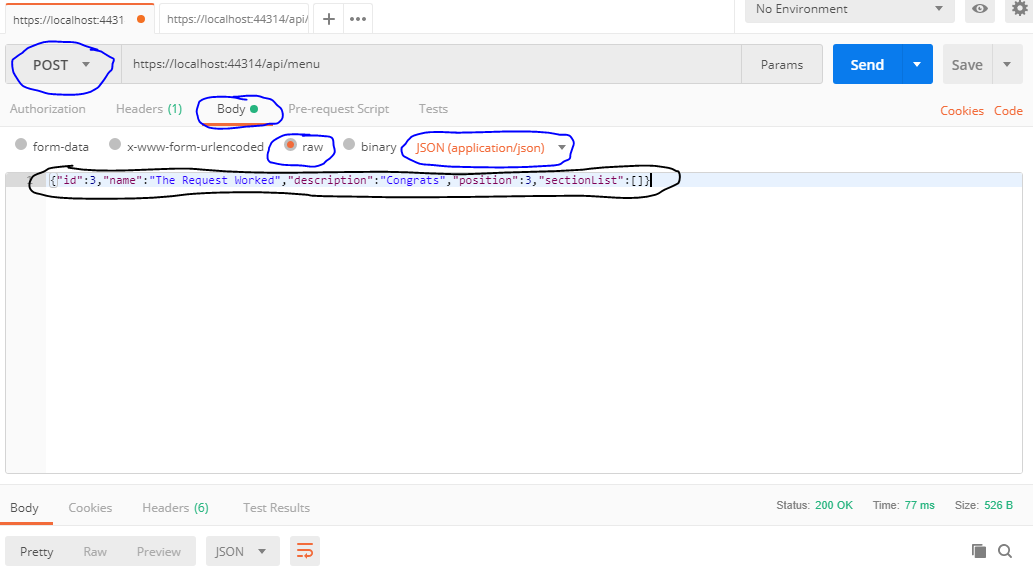


If the request is successful it will return a response with status code 200 (OK) and a body of JSON that should show all of the menus currently in the database.

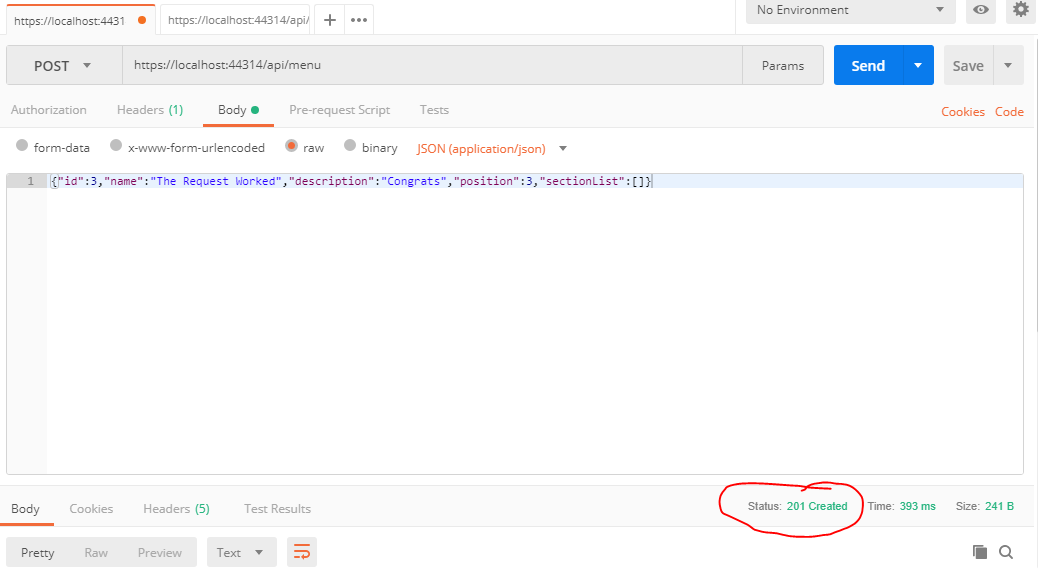
**NOTE**: That for these queries, yours may not return exactly the same data as in our screenshots, what’s important is that it returned the kind of response you were expecting.

Now try a different request. Let’s try a POST request. Change the request from GET to POST. Select the Body Tab. Select the raw bubble. Then change the body type from text to application/json. Then paste in this JSON:

{"id":3,"name":"The Request Worked","description":"Congrats","position":3,"sectionList":[]}

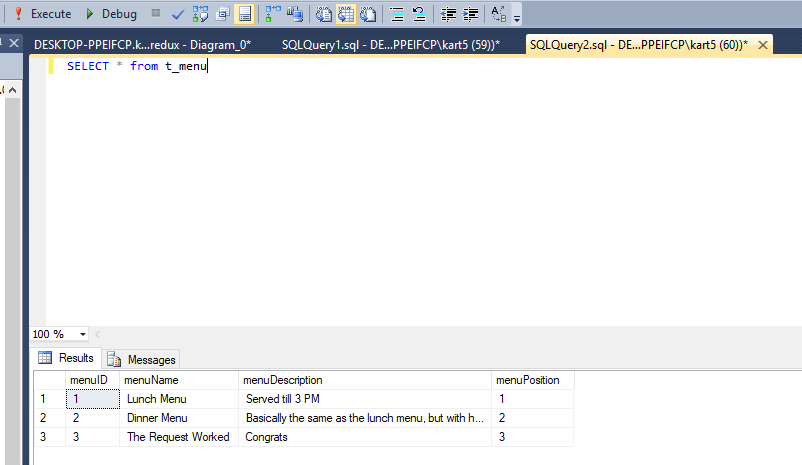


Now click send and see what response you get back. A status code of 201 (content created) will indicate success and 500 (server error), failure.



Go back into SQL management studio and run this query:

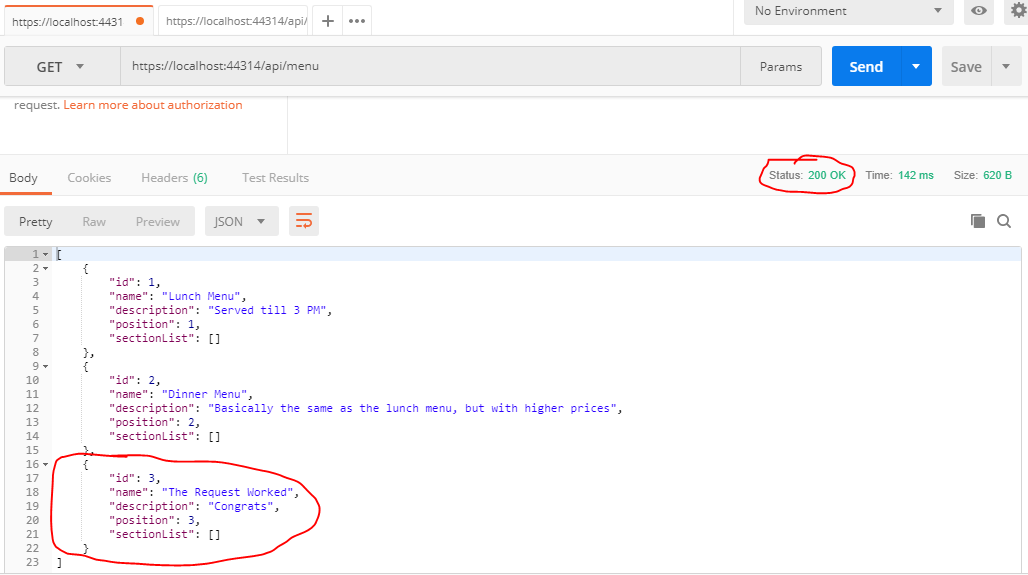
SELECT \* from t\_menu



You should see the new menu that we added using POSTMAN.

Try finding that menu in POSTMAN. Change the HTTP method from POST back to GET.

It should appear at the bottom. If so: Congrats! Your API works.



**9.2 Provided POSTMAN Request Collection**

## The project files include a premade selection of POSTMAN requests that can test every functionality of the API.

Go to File 🡪 import 🡪 Choose Files and select KaibaReduxPostmanRequestCollection.json from the working KaibaReduxAPI folder.

## 

## This will open a collection of requests. Click on the KaibaRedux folder and its child folders to view all the provided requests. Click on a request to bring it up. The first request is exactly the same as the first one from the last section. Try Running it.

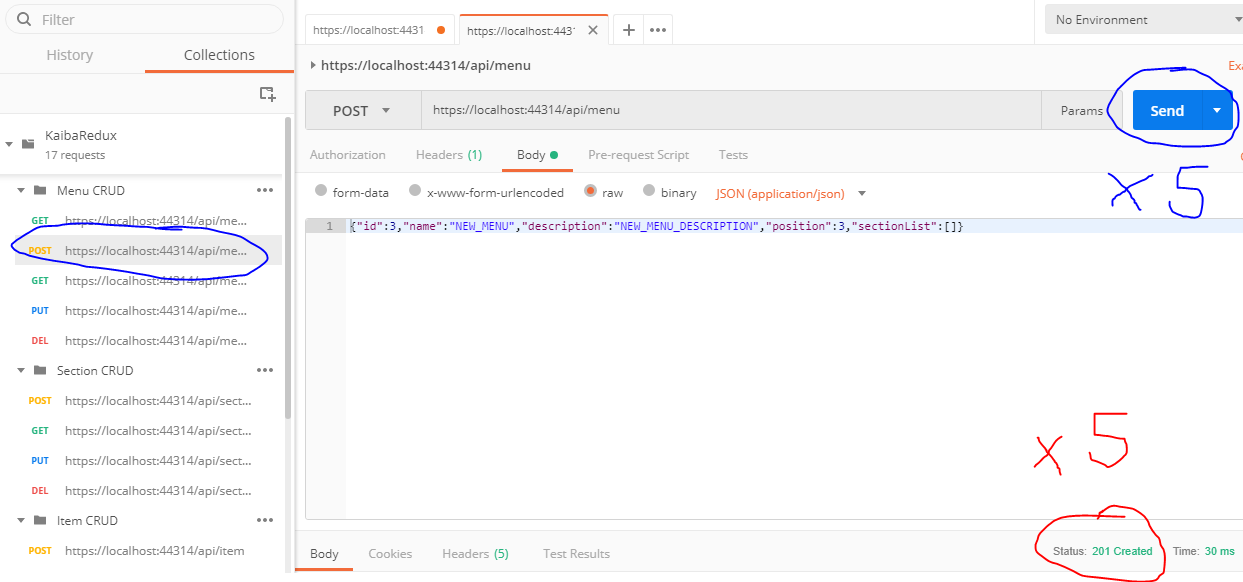
## 

## Run the DB reset SQL statements again to get rid of any changed data.

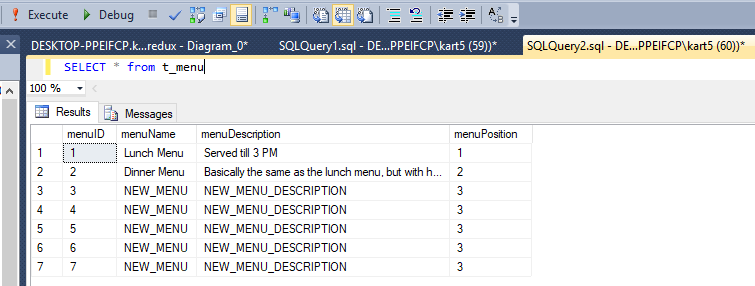
## Now try running each request from top to bottom. If they all succeed (ei. none return a 500 or 404 error code) and all return 200, 201 or 204, then your API is fully functional.

Try Running and modifying different requests. For example, try changing the position or menuID key id of some sections, and see how that changes the database by running SELECT statements in SQL Management Studio.

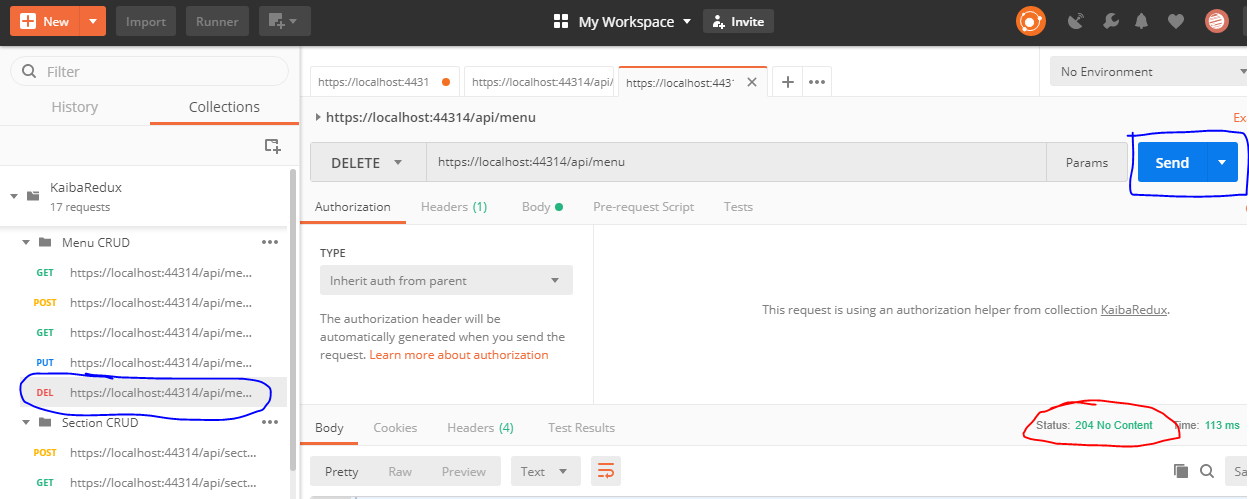
**NOTE**: Because of how the database is set up, the IDs of each new record are automatically assigned and cannot be specified. This means that some of the requests will fail if run out of order or multiple times. For example, trying running the POST menu request a four or five times. It should succeed (201 code) each time.



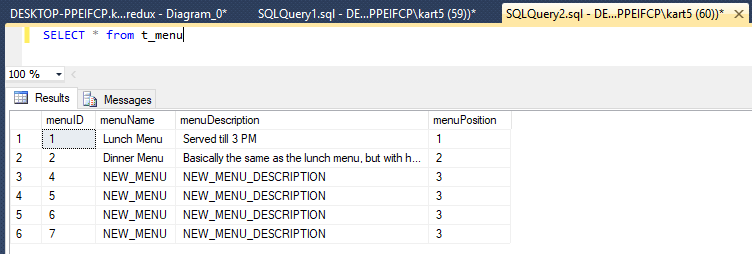
Now run SELECT \* from t\_menu in SQL Management Studio. Notice how each new menu is given a different menuID.



Now try running the DELETE menu request. Should succeed with 204.

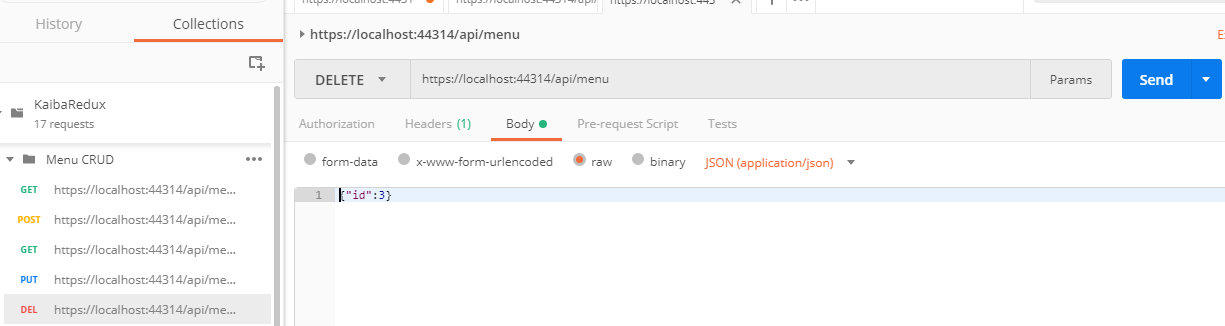


Now Run the SELECT \* from t\_menu in SQL Management Studio again.

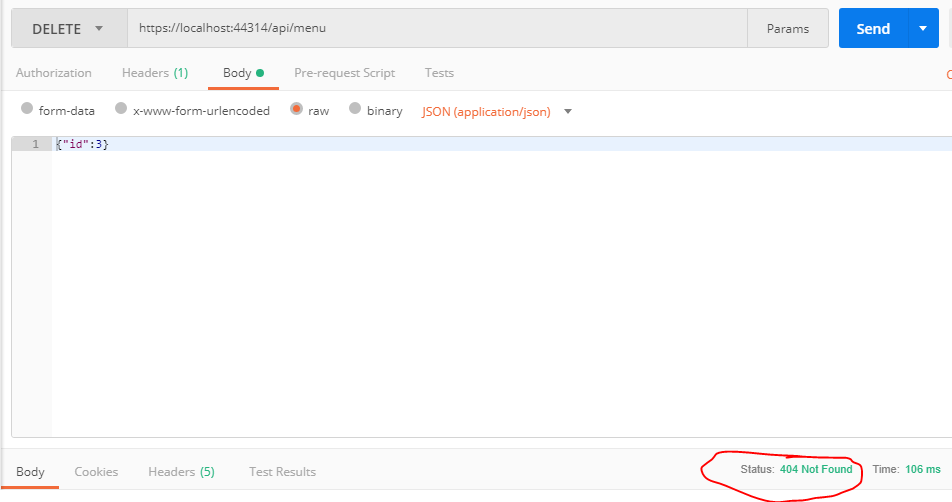


As expected the menu with ID = 3 is gone (the menuID column, not the leftmost column)

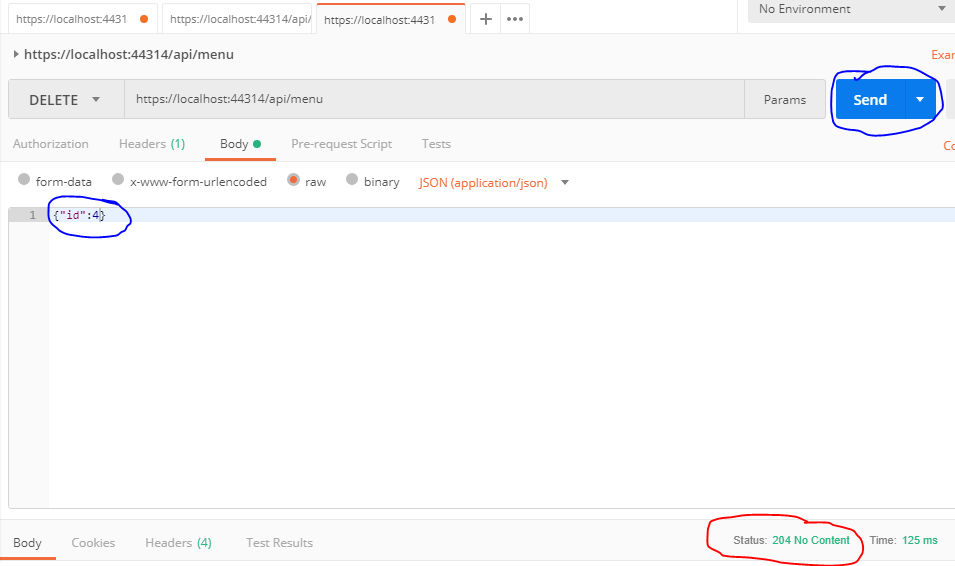
Now bring up the DELETE menu request again. Go to the Body tab and check the JSON there.



This JSON is specifying to delete the menu with ID = 3, but we’ve already done that. Try sending the delete request, and see that it fails with a 404 error code.

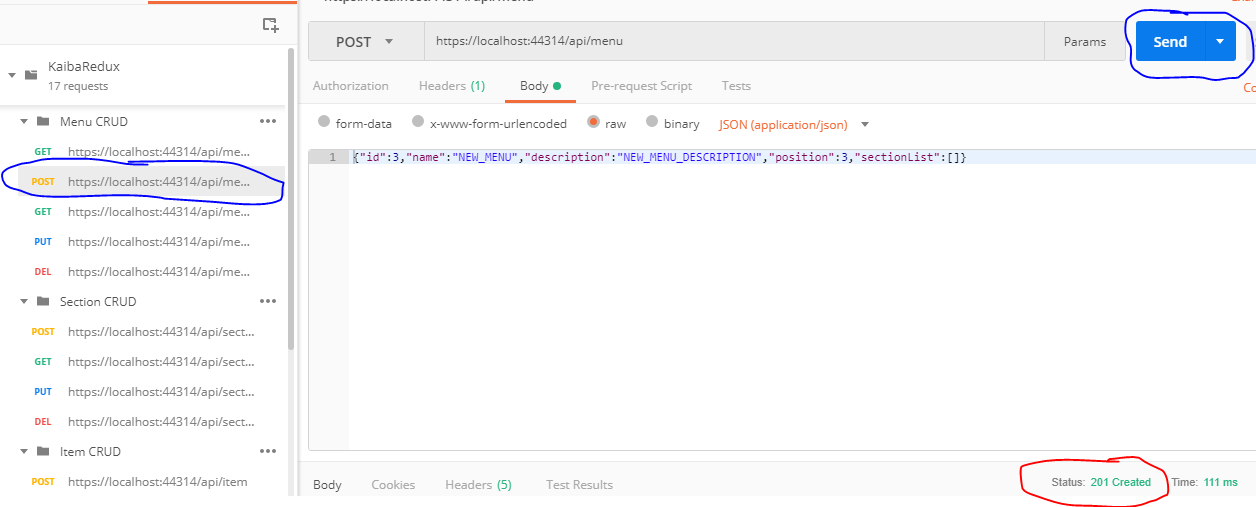


Now change in the JSON change 3 to 4. And send this new DELETE request.

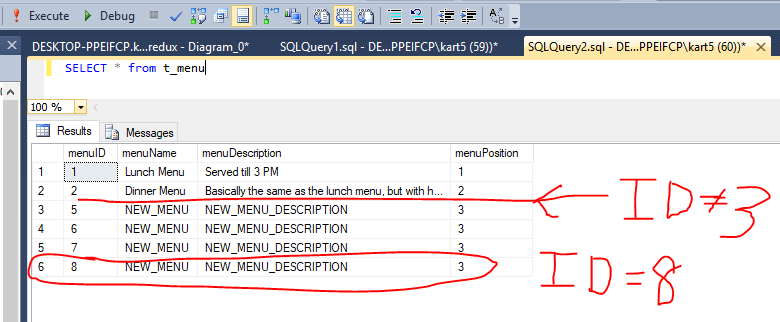


It succeeds with 204, because that record exists. Try running it again, and it fails.

You might think you to fix this by adding another menu, so try running the POST menu request.



Then check run the SELECT \* from t\_menu SQL query in Management Studio again.



Notice how the latest record was added with ID = 8 and not ID = 3. This will is why running the delete statements multiple times will fail, unless you change the ID specified in the JSON. Similarly you can change the ID specified in the GET and PUT requests.

Running the SQL reset statements will reset the database ID autoincrement, so that the original POSTMAN requests will work again

## **10 The Client**

## **10.1 Introduction to HTML, JavaScript, jQuery, and Ajax**

HTML is the language in which web pages are written. It consists of pairs of tags like:

<p> This is a Paragraph tag </p>

If you need more instruction in HTML see this tutorial: <https://www.w3schools.com/html/html_intro.asp>

JavaScript is a scripting language that is written within web pages and runs in the browser while the page is open. It allows you to add, change, and delete HTML elements while the page is open

If you need more instruction in JS see this tutorial:

<https://www.w3schools.com/js/js_intro.asp>

**Note**: The W3Schools tutorials are some of the best and are available for free. Make sure you click the “Try it Yourself” Links because they bring up a wonderful embedded IDE and interpreter, allowing you to make changes to the code real time and see exactly what the effects are.

jQuery is an extremely useful JS library that makes working with dynamic HTML elements much easier.

As before, see W3 for more: <https://www.w3schools.com/Jquery/jquery_intro.asp>

Ajax is a way of having a browser send or receive data from a server without having to reload the page. That data can be either XML or JSON (which is what we’ll be using). It can be fairly complex, but we will be using jQuery’s simplified Ajax functionality.

See <https://www.w3schools.com/Jquery/jquery_ajax_intro.asp>

**10.2 Project Configurations**

## In Visual Studio in the solution explorer on the right (not visible while the project is running), open the Startup.cs file, which contains several configuration settings.

## In the Configure method add these two lines before the app.UseMvc(); line.

app.UseDefaultFiles();

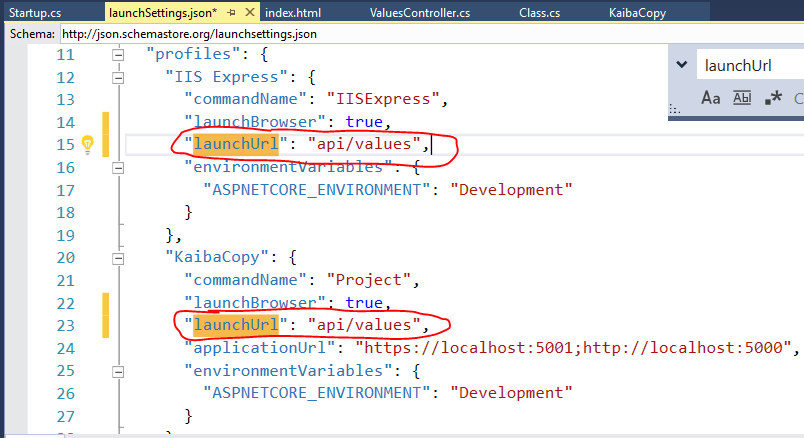
## app.UseStaticFiles();

## 

## This allows us to use plain old normal URL routing.

In the solution explorer, open the properties folder (click small triangle on left), and open launchsettings.json. This file contains configurations for what ports to use and what default launch URL to use.

Find both launchURL attributes and delete the whole line, including the comma at the end.

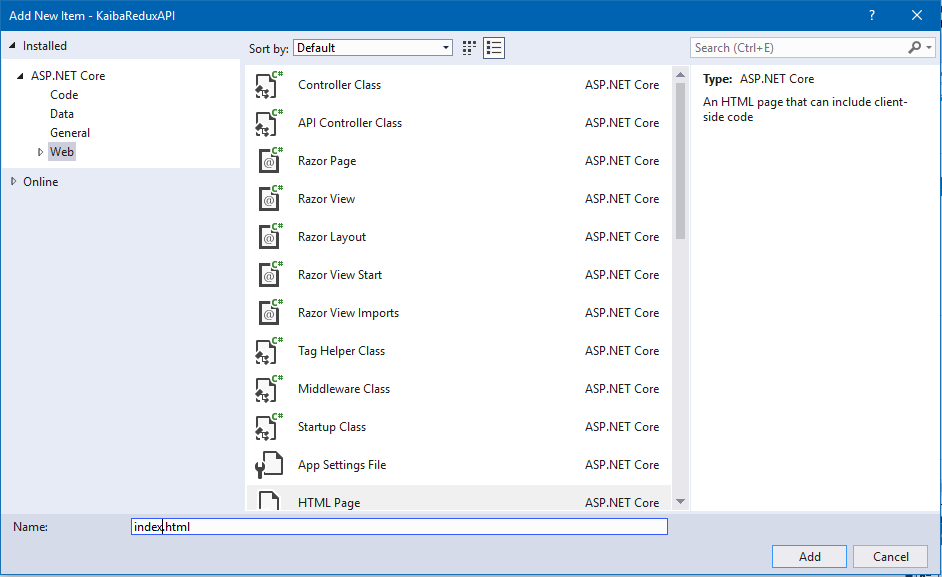


## This will set the starting launch URL to the default, which we’ll set up in the next section.

## **10.3 The homepage: index.html**

In most web servers the default homepage is index.html (or .asp, .php, .etc). In ASP.NET the wwwroot folder contains our webpages, so that’s where we will put index.html.

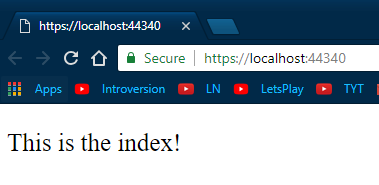
In Visual Studio in the solution explorer on the right (not visible while the project is running), right click wwwroot 🡪 add 🡪 add item, select web in the top left, and scroll down to HTML Page. Name it index.html



The index.html page is very empty. Add this line in the body section:

<p> This is the index!</p>

Run the project. If you did the configuration in the previous section correctly, then the index.html page should open in your browser. You should see This is the index! on the page.



Replace the index.html code (note that HTML and JS files can be edited while the project is running, allowing you to make quick changes, then just refresh the page to see the effect).

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Kaiba Redux Menu</title>

<!-- Get the Jquery library from google-->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Load the separate JavaScript file-->

<script src="js/JavascriptFunctions.js"></script>

<script>

// This is the javascript for this page

// Set id of current menu being shown

// taken from id url parameter, if not set defaults to 1

if (getQueryParam("id") == null) {

var currentMenuID = 1;

}

else {

// otherwise load the info for the menu that is being edited

var currentMenuID = parseInt(getQueryParam("id"));

}

// global to denote whether edit buttons are hidden or not

// set to true to have the buttons default to being shown

var editButtonsShowing = false;

$(document).ready(function () {

// code written here is not executed until the web page has finished loading, preventing any errors with page elements not having loaded yet

// get the various menus from the API

getMenus();

// show the first menu (defaults to currentMenuID's value)

showMenu(currentMenuID);

// assign show/hide edit events to the login button

$("#loginButton").click(function () {

if (editButtonsShowing) {

hideEditButtons();

editButtonsShowing = false;

}

else {

showEditButtons();

editButtonsShowing = true;

}

});

});

</script>

</head>

<body>

<h1>Menu</h1>

<button id="loginButton"> Login </button>

<br />

<span id="loginDesc"> <i> Press to show edit buttons </i> </span>

<br />

<br />

<p> List of Menus</p>

<p><i> (Click to show that menu) </i></p>

<div id="menuList"></div>

<br />

<br />

<p><i> Menu Contents </i></p>

<div id="menuContents"></div>

<br />

<br />

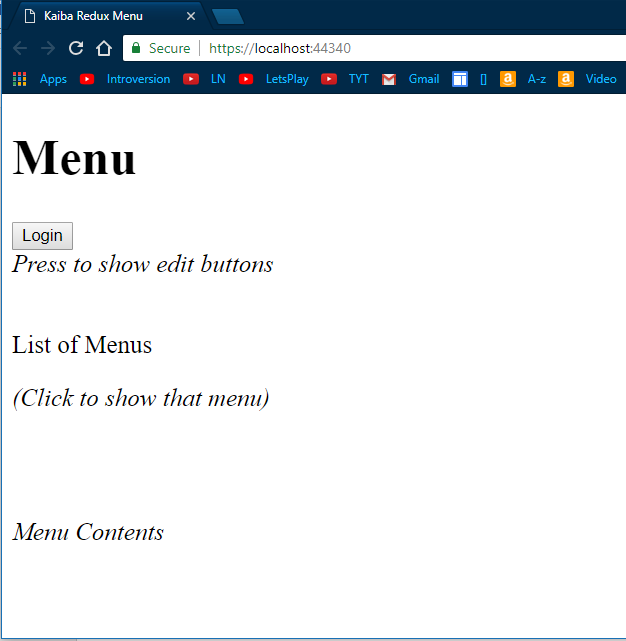
<br />

<br />

</body>

</html>

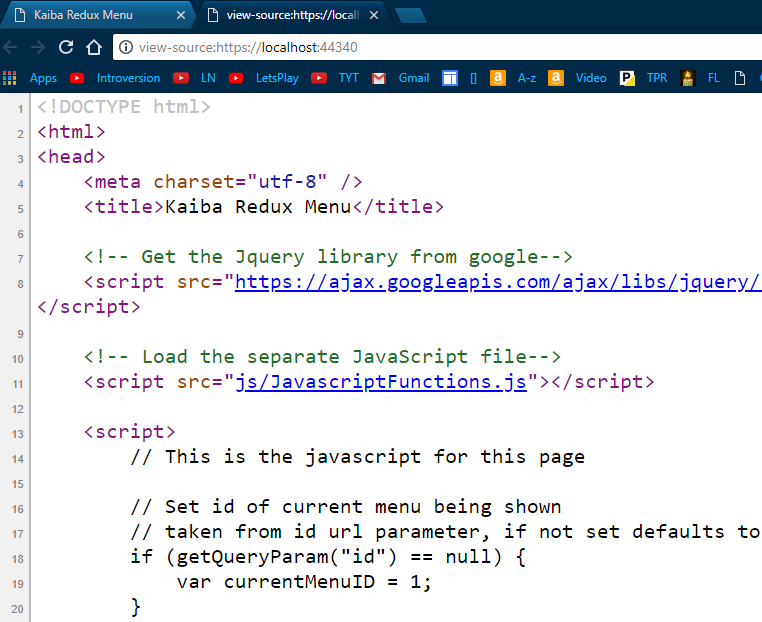
Run the project (or if it’s already running refresh the index.html page)



## **10.4 Using Your Browser’s Developer’s Console**

These directions are geared towards using Google Chrome. You can use any major browser (except Internet Explorer), but some instructions may be slightly different, and some screens may look different. We recommend using Google Chrome as your default browser, while working on this project. But the choice is up to you (Mozilla Firefox is also good).

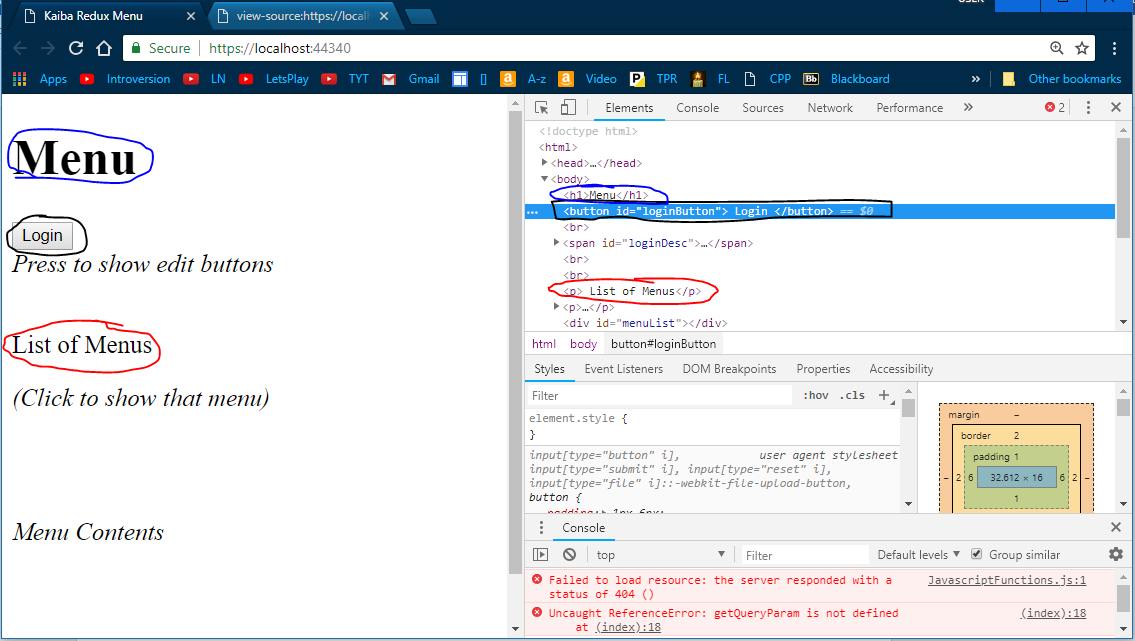
Bring up the index.html page from the last section. Right click anywhere on the page and select view page source (if using Chrome) or view source code (for other browsers). This will bring up the HTML that defines what the page looks like.



This source code will always be exactly what is written in the index.html file.

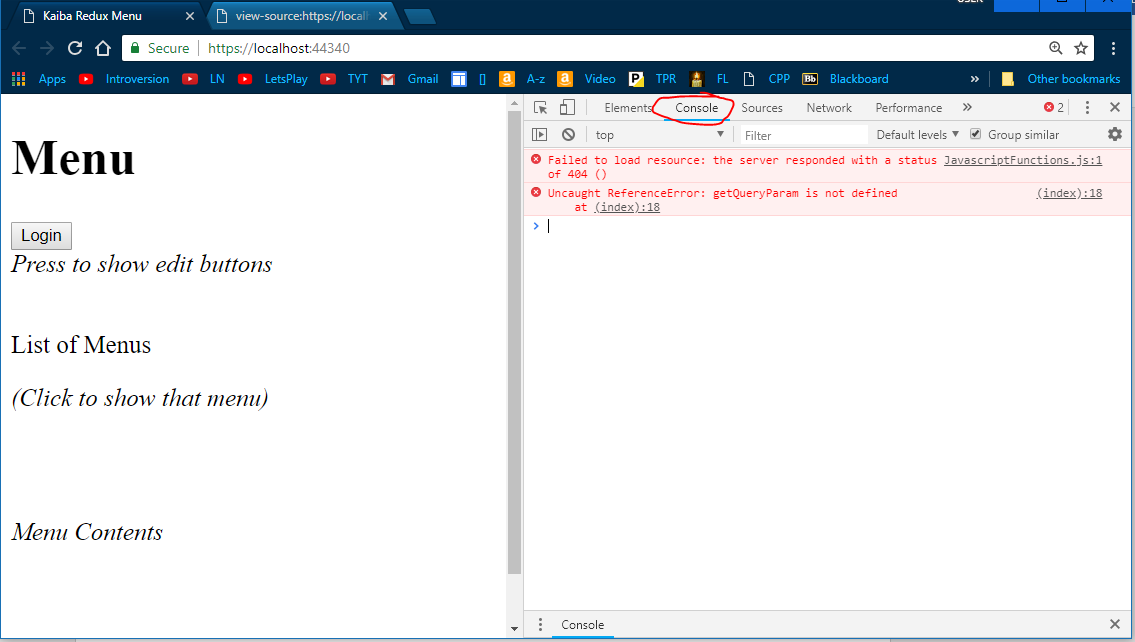
However, this project will be using a lot of dynamic HTML elements, which will not appear in the source code. To see the HTML that is currently being displayed on the page, we use inspect.

On the index.html page, right click on the login button and select inspect (Chrome) or inspect element (some others).



This will bring up the developer’s console, which allows you to see exactly what’s on this webpage. In the upper right, you can see the HTML that corresponds to the elements currently on the webpage. Right clicking and selecting inspect on any element will show you the HTML statements that correspond to that element. This is very useful when trying to troubleshoot and debug the webpage.

In the lower right, you might notice that some errors have been thrown. Click on Console in the upper right. This opens the console, which shows any JS errors that have occurred.



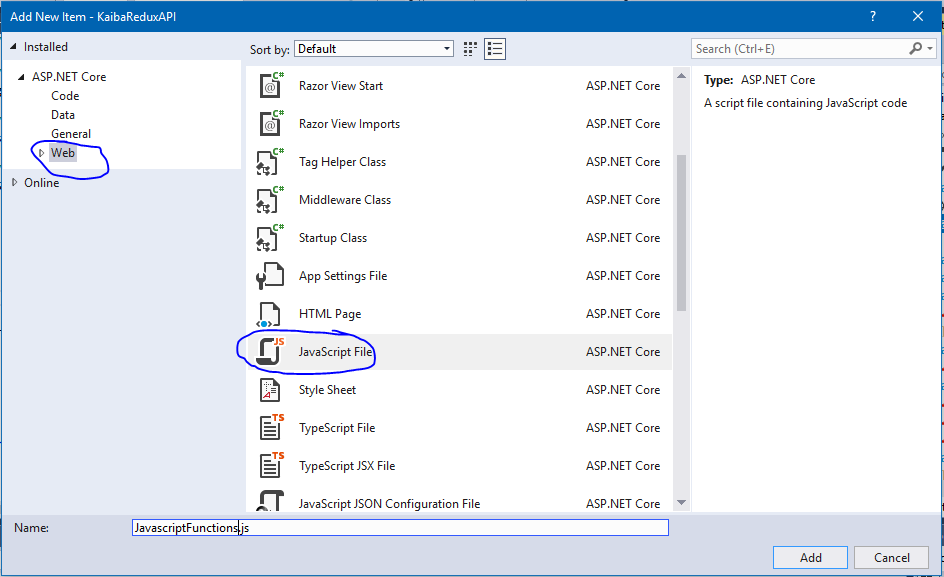
The first error says that it couldn’t find the JavascriptFunctions file, which is expected as we haven’t created it yet. The second error is about a function that isn’t defined yet. Creating the JavascriptFunctions.js file will fix both errors.

**Note**: That clicking on the link at the right of the error message will take you to the sources section, where you can figure out exactly where the error was thrown. Also note that you can find the line number. For example, the second error occurred on line 18 of the index.html page. The developer’s console is very powerful, but beyond the scope of this guide.

## **10.5 The JavaScript Functions**

In the solution explorer, right click the wwwroot folder, and select new folder. Name it “js” It is a programming convention to store all JavaScript files inside a folder called js.

Right click the new js folder, select Add 🡪 Add Item. Under Web, Scroll down to JavaScript File. Name it JavascriptFunctions.js



This file will contain all of the JavaScript Functions.

Add the constants, which are values that will be used throughout the file.

// This url should point to API

const URL = "api/";

// The id of the div in which to put the elements describing all the menus

const MENUS\_LIST\_DIV\_ID = "#menuList";

// The id of the div in which to put the elements describing the selected menu's contents

const MENU\_CONTENTS\_DIV\_ID = "#menuContents";

// The id of the div on the editItem page that contains the pricelines

const PRICELINE\_DIV\_ID = "#pricelineDiv";

Let’s add the utility JS functions:

First getQueryParam(), which takes a string and checks the URL for a URL parameter matching the given string, then returns that parameter’s value. Returns null if no such parameter.

function getQueryParam(param) {

// searches the url parameters for a specified parameter, which it returns if it exists

// returns null if the parameter is not found

let result = null;

location.search.substr(1)

.split("&")

.some(function (item) { // returns first occurence and stops

return item.split("=")[0] == param && (result = item.split("=")[1])

})

return result

}

For more info on URL parameters see this StackOverflow discussion:

<https://stackoverflow.com/questions/39266970/what-is-the-difference-between-url-parameters-and-query-strings>

Create the showEditButtons() and hideEditButtons(). These functions will hide and show the edit and delete buttons. It also changes the text of the login button.

function showEditButtons() {

// shows all edit buttons (which start hidden)

$("#loginButton").text("Logout");

$("#loginDesc").text("Press to hide edit buttons");

$(".edit").show();

$(".delete").show();

}

function hideEditButtons() {

// shows all edit buttons (which start hidden)

$("#loginButton").text("Login");

$("#loginDesc").text("Press to show edit buttons");

$(".edit").hide();

$(".delete").hide();

}

Lastly isPositiveNumber(), which will be used for data validation and error message creation. Takes a value and a string. First it checks that the value is not null, is not empty string, is a number, and that it’s positive. If all those conditions are true, then it returns empty string (no error message). Otherwise it returns an error message, using the given string.

function isPositiveNumber(input, fieldName) {

// takes an input and checks whether it's a positive number or not

// second param is the name of the field being checked (used in error message)

// returns empty string if true, an error message string if false

let result = "";

// check that's it's not null or an empty string, a number and greater than 0

// isNaN() returns true if the input is not a number, false if it is

// so get the opposite using not (!)

let isNum = (input != null) && (input != "") && (!isNaN(input)) && parseFloat(input) > 0;

if (isNum) {

result = "";

}

else {

result = "" + fieldName + " must be a positive number. \n";

}

return result;

}

**Note**: JavaScript is a weakly-typed language, which means that you do not have to specify the data type of variables when you declare them. Essentially that means that is can be difficult to know exactly what kind of data a variable holds, hence the careful checking. It also means that JS comparisons can have some unusual results (caused by auto type casting ). For example,

"" == 0 // yields true, because somehow empty string is equivalent to 0

"" == [] // yields true, because somehow empty string is also equivalent to empty array

See here for more on weakly typed vs strongly typed languages: <https://en.hexlet.io/courses/intro_to_programming/lessons/types/theory_unit>

If you’re already an experienced programmer, then this might be the most hilarious thing ever

(Warning, if you are not a programmer, then this will make no sense and be boring):

<https://www.destroyallsoftware.com/talks/wat>

## **10.6 jQuery features**

If you are already familiar with jQuery and callback functions, then feel free to skip this section.

Consider the head section of index.html (reference only, don’t paste it anywhere)

<head>

<meta charset="utf-8" />

<title>Kaiba Redux Menu</title>

<!-- Get the Jquery library from google-->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Load the separate JavaScript file-->

<script src="js/JavascriptFunctions.js"></script>

<script>

// This is the javascript for this page

// Set id of current menu being shown

// taken from id url parameter, if not set defaults to 1

if (getQueryParam("id") == null) {

var currentMenuID = 1;

}

else {

// otherwise load the info for the menu that is being edited

var currentMenuID = parseInt(getQueryParam("id"));

}

// global to denote whether edit buttons are hidden or not

// set to true to have the buttons default to being shown

var editButtonsShowing = true;

$(document).ready(function () {

// code written here is not executed until the web page has finished loading, preventing any errors with page elements not having loaded yet

// get the various menus from the API

getMenus();

// show the first menu (defaults to currentMenuID's value)

showMenu(currentMenuID);

// assign show/hide edit events to the login button

$("#loginButton").click(function () {

if (editButtonsShowing) {

hideEditButtons();

editButtonsShowing = false;

}

else {

showEditButtons();

editButtonsShowing = true;

}

});

});

</script>

</head>

The first <script src="SomeURL"></script> call gets the jQuery library from a Google API. The second loads in all the functions contained in the JavaScriptFunctions.js file. The first call must always go before the second, because those functions require jQuery.

The $(document).ready(function () { … } part contains code that will be executed after the page has completely finished loading. Any code that effect any HTML elements should be put here, otherwise those elements might not have been loaded yet.

Consider this js snippet:

$("#loginButton").click(function () {

if (editButtonsShowing) {

hideEditButtons();

editButtonsShowing = false;

}

else {

showEditButtons();

editButtonsShowing = true;

}

});

This shows several jQuery features.

* $() is how you invoke jQuery

## $("#loginButton") is how you select the HTML element with id = “loginButton”

## $("SelectionString").click(function () { … }) is how you assign an onclick function

## The callback function provided will be called whenever that element is clicked

Consider the showEditButtons() function again

function showEditButtons() {

// shows all edit buttons (which start hidden)

$("#loginButton").text("Logout");

$("#loginDesc").text("Press to hide edit buttons");

$(".edit").show();

$(".delete").show();

}

* $("SelectionString").text("Logout"); Changes the text of an element to “logout”
* $(".edit") is how you select all HTML elements of the “edit” class
  + (all HTML elements with class = "edit")
* $("SelectionString").show(); and .hide() make an element visible and invisible

## For more on jQuery selection strings: <https://www.w3schools.com/Jquery/jquery_selectors.asp> **10.7 Dynamic JavaScript Functions: getMenus()**

Index.html requires 2 js functions, the first is getMenus(). Inside JavascriptFunctions.js add:

function getMenus() {

// This function gets the list of menus from the API

// It displays them as <p> elements within the div given by constant MENUS\_LIST\_DIV\_ID

// It sets the onclick event of each to call showMenu(#), where # is that menu's id number

// Jquery ajax call to get the list containing the menu info

$.ajax({ // The $ (dollar sign) is used to access the Jquery functions, in this case an ajax call

method: 'GET', // This is a GET request

url: URL + "menu", // The URL we want is api/menu

dataType: "json", // The datatype we expect the server to return, the JSON will automatically be parsed and converted into a JavaScript object

success: function (menuList) { // This inline function will be called if the request is successful

// Note that this function is called asynchronously at a later time, so any data you get back must be maniputed here.

// the variable 'menuList' is now a js array that holds menu objects

// It cannot be used outside of this function

// Use Jquery to empty the menuList div of any elements

$(MENUS\_LIST\_DIV\_ID).empty();

// loop through them

for (let i = 0; i < menuList.length; i++) {

let menu = menuList[i];

// add line break if not the first menu

if (i > 0) {

$('<br />').appendTo($(MENUS\_LIST\_DIV\_ID));

}

// Using Jquery create a new <p> object with the menu's info

// It has id property: id = menu#, where # is that menu's id number

// It also adds an onclick event, so that when clicked it will call showMenu(#), where # is that menu's id number

$('<button id="menu' + menu.id + '" onclick="showMenu(' + menu.id + ')"><strong> Show ' + menu.name + '</strong></button>' +

'<button class="edit" onclick="editMenu(' + menu.id + ')"> Edit </button>' + // adds a button to go to the menu edit page

'<button id="deleteMenu' + menu.id + '" class="delete" onclick="deleteMenu(' + menu.id + ')"> Delete </button>' // adds a button to delete the menu

).appendTo($(MENUS\_LIST\_DIV\_ID)); // Add it to the menuList div

}

// add a create new menu button

$('<br /> <button class="edit" onclick="editMenu()"> Create new Menu </button>').appendTo($(MENUS\_LIST\_DIV\_ID));

},

error: function (jqXHR, status, errorThrown) { // This function will run if there's an error

alert("ERROR: Can't retrieve menu list " + errorThrown + " "); // Pop up a textbox with an error message

}

});

}

Notice the structure of the jQuery ajax call (don’t paste it anywhere):

$.ajax({ // This is a jQuery ajax call

method: 'GET', // This is a GET request

url: URL + "menu", // The URL we want is api/menu

dataType: "json", // The datatype we expect the server to return

success: function (menuList) { … },

error: function (jqXHR, status, errorThrown) { … }

});

## The method can be GET, POST, PUT, or DELETE (there are more, but not in this project)

## The url tells the browser where to send the request

## This should be the same URLs that the POSTMAN requests have

## dataType: just tells the browser that we’re expecting to receive JSON back

## success: function (menuList) { … } defines a callback function to be called once the data has been received.

## Since the request will take time, we have to wait for the response

## This is called an asynchronous function, because it is called at a later time, not immediately after the previous statement as in usual program flow

## The menuList object will now be a JS object that was automatically created from the JSON that was recieved

## error: function (jqXHR, status, errorThrown) { … } defines a callback function for when a request fails

## This is also an asynchronous function

Consider this jQuery statement and its comments (don’t paste it anywhere):

$(

// Here we create a string that contains several HTML elements

// The first element is a button with id="menu#", where # is the menuID

'<button id="menu' + menu.id +

// when clicked it triggers a call to showMenu(#), where # is the menuID

'" onclick="showMenu(' + menu.id + ')">' +

// It’s text is bolded and says “Show NAME”, where NAME is the menu’s name

'<strong> Show ' + menu.name + '</strong></button>' +

// Here is another button, it’s class = “edit” and when clicked it calls editMenu(#)

'<button class="edit" onclick="editMenu(' + menu.id + ')"> Edit </button>' +

// Here is a final button, it has id = “deleteMenu#”, so that we can find it later

// it’s class = “delete” and when clicked it calls deleteMenu(#)

'<button id="deleteMenu' + menu.id + '" class="delete" onclick="deleteMenu(' + menu.id + ')"> Delete </button>'

// Lastly we take that whole HTML string and put it inside the MenuList Div.

).appendTo($(MENUS\_LIST\_DIV\_ID));

## **10.8 Dynamic JavaScript Functions showObject() Methods**

The ShowMenu() function takes a menuID and displays the contents of that menu within the MenuContents div. It calls the ShowSection(), ShowItem(), and ShowPriceline() functions that take their respective objects and then add HTML elements describing them to the MenuContents div.

Add the following to JavascriptFunctions.js:

function showMenu(id = 1) {

// shows a specific menu, the first paramenter is the id, which defaults to the current menu being shown

// The affected div is given by the MENU\_CONTENTS\_DIV\_ID constant

// creates nested divs and <p> inside the specified div (also clears any elements currently in the div)

// Jquery ajax call to get the specific menu

$.ajax({

method: 'GET',

url: URL + "menu/" + id, // The URL we want is api/menu/#

dataType: "json",

success: function (menu) { // function called if successful

// Note that this function is called asynchronously at a later time, so any data you get back must be maniputed here.

// the variable 'menu' is now a menu object that cannot be used outside of this function

// Use Jquery to empty the div of any elements

$(MENU\_CONTENTS\_DIV\_ID).empty();

// Using Jquery create a new <p> elements with the menu's info

$('<p id="menu' + menu.id + '" ><strong>' + menu.name + '</strong></p>' +

'<p><i> ' + menu.description + ' </i></p>'

).appendTo($(MENU\_CONTENTS\_DIV\_ID)); // Add it to the specified div

// add a create new section button, passes a negative id, which signifies that it is a create, the negative number is the menuId to within which the section should be created

$('<button class="edit" onclick="editSection(-' + menu.id + ')"> Create new Section in ' + menu.name + '</button> <br /> <br />').appendTo($(MENU\_CONTENTS\_DIV\_ID));

// loop through the sectionList variable of the menu object

for (let i = 0; i < menu.sectionList.length; i++) {

let section = menu.sectionList[i];

// display each section

showSection(section);

}

if (menu.sectionList.length > 0) {

// add a create new section button

$('<br /> <button class="edit" onclick="editSection(-' + menu.id + ')"> Create new Section in ' + menu.name + '</button>').appendTo($(MENU\_CONTENTS\_DIV\_ID));

}

// change id of the current menu being shown

currentMenuID = id;

// check if edit buttons should be hidden

if (editButtonsShowing == false) {

hideEditButtons();

console.log("hide");

}

},

error: function (jqXHR, status, errorThrown) { // This function will run if there's an error

alert("ERROR: Can't retrieve that menu " + errorThrown + " "); // Pop up a textbox with an error message

}

});

}

function showSection(section) {

// takes a section object and adds <p> describing it to the menu contents div (given by the constant MENU\_CONTENTS\_DIV\_ID)

$('<p id="section' + section.id + '" ><strong>' + section.name + '</strong></p>' +

'<p><i> ' + section.description + ' </i></p>' +

'<button class="edit" onclick="editSection(' + section.id + ')"> Edit ' + section.name + '</button>' + // adds a button to go to the section edit page

'<button id="deleteSection' + section.id + '" class="delete" onclick="deleteSection(' + section.id + ')"> Delete ' + section.name + '</button>' + // adds a button to delete the section

'<br />'

).appendTo($(MENU\_CONTENTS\_DIV\_ID)); // Add it to the specified div

// add a create new item button

$('<button class="edit" onclick="editItem(-' + section.id + ', ' + section.menuID + ')"> Create new Item in ' + section.name + '</button> <br /> <br />').appendTo($(MENU\_CONTENTS\_DIV\_ID));

// loop through the itemList variable of the section object

for (let i = 0; i < section.itemList.length; i++) {

let item = section.itemList[i];

// display each item

showItem(item, section.menuID);

}

}

function showItem(item, menuID = 1) {

// takes an item object and adds <p> describing it to the menu contents div (given by the constant MENU\_CONTENTS\_DIV\_ID)

// second parameter is the menuID of the section that this menu is in, which defaults to 1 if not specified

$('<p id="item' + item.id + '" >' + item.position + '. ' + item.name + '</p>' +

'<p><i> ' + item.description + ' </i></p>' +

'<button class="edit" onclick="editItem(' + item.id + ', ' + menuID + ')"> Edit ' + item.name + '</button>' + // adds a button to go to the item edit page

'<button id="deleteItem' + item.id + '" class="delete" onclick="deleteItem(' + item.id + ')"> Delete ' + item.name + '</button>' // adds a button to delete the item

).appendTo($(MENU\_CONTENTS\_DIV\_ID)); // Add it to the specified div

// check if there's only one priceline item with no description

if (item.priceLineList.length == 1 && item.priceLineList[0].description == "") {

// append the price to the end of the item name <p>

$('#item' + item.id).append($('<span> - $' + item.priceLineList[0].price + '</span>'));

}

else {

// otherwise loop through the priceLineList variable of the item object

for (let i = 0; i < item.priceLineList.length; i++) {

let priceline = item.priceLineList[i];

// display each item

showPriceline(priceline);

}

}

// add line break

$('<br />').appendTo($(MENU\_CONTENTS\_DIV\_ID));

}

function showPriceline(priceline) {

// takes an priceline object and adds <p> describing it to the menu contents div (given by the constant MENU\_CONTENTS\_DIV\_ID)

$('<p id="priceline' + priceline.id + '" >' + priceline.description + ' - $' + priceline.price + '</p>'

).appendTo($(MENU\_CONTENTS\_DIV\_ID)); // Add it to the specified div

}

Notice how similar the showSection() and showItem() functions are to each other. Their basic structure is this (described specifically for showSection(), names differ for showItem() )

* Receive a section object
* Create a few <p> elements describing it’s name and description
* Create an edit button and a delete button, each with the appropriate class and onclick function
* Create a “create new item” button, which takes us to a new page to create a new item
* Loop through your itemList calling showItem() for each

The index.html page should now show specific menus when clicking on the show menu buttons. The edit, delete, and create buttons will not work however.

## **10.9 Menu: Create, edit, and delete**

Under wwwroot, add a new HTML item, name it editMenu.html. Replace code with:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Edit Menu</title>

<!-- Get the Jquery library from google-->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Load the separate JavaScript file-->

<script src="js/JavascriptFunctions.js"></script>

<script>

// This is the javascript for this page

// global variable denoting whether this is a create or edit

var create = false;

$(document).ready(function () {

// code written here is not executed until the web page has finished loading, preventing any errors with page elements not having loaded yet

// change heading to Create menu if that's what we're doing

if (getQueryParam("id") == null) {

create = true;

$('#heading').text("Create Menu Page");

$('#id').hide();

$('#id').val("0");

$('#idLabel').hide();

}

else {

// otherwise load the info for the menu that is being edited

loadMenu(getQueryParam("id"));

}

// set onclick event for the submit button

$('#submit').click(function () {submitMenu(create) });

});

</script>

</head>

<body>

<h2 id="heading"> Edit Menu Page</h2>

<div>

<span id="idLabel">ID: </span><br>

<input type="text" id="id" value="" readonly>

<br>

Menu Name:<br>

<input type="text" id="name" value="">

<br>

Menu Description:<br>

<input type="text" id="description" value="">

<br>

Menu Position:<br>

<input type="text" id="position" value="">

<br><br>

<input type="submit" value="Submit" id="submit">

</div>

<br><br>

<a id="menuDisplayLink" href="index.html">Back to Menu Display</a>

</body>

</html>

Notice how it uses getQueryParam() function to check for an id parameter. If it finds one, this is an update and it calls loadMenu(id). Otherwise it leaves the fields blank, because this is a create operation.

Add following to JavascriptFunctions.js:

function editMenu(id = null) {

// redirects to the Menu Edit/Create page, passes an id as a url parameter if one is passed

let param = "";

if (id != null) {

param = "?id=" + id;

}

window.location = "editMenu.html" + param;

}

function loadMenu(id) {

// takes a menu id and loads it's info into the input elements (id, name, description, and position)

// Jquery ajax call to get the specific menu

$.ajax({

method: 'GET',

url: URL + "menu/" + id, // The URL we want is api/menu/#

dataType: "json",

success: function (menu) { // function called if successful

$('#id').val(menu.id);

$('#name').val(menu.name);

$('#description').val(menu.description);

$('#position').val(menu.position);

},

error: function (jqXHR, status, errorThrown) { // This function will run if there's an error

alert("ERROR: Can't retrieve that menu " + errorThrown + " "); // Pop up a textbox with an error message

}

});

}

function submitMenu(create) {

// takes the values of the menu input elements (id, name, description, and position)

// and passes them to the api using create if true is passed and using edit if false is passed

// data validation: check if position is a number

let errorString = isPositiveNumber($('#position').val(),"Position");

if (errorString == "") {

// create menu object by getting value attribute of the input elements

let menu = {

"id": parseInt($('#id').val()),

"name": $('#name').val(),

"description": $('#description').val(),

"position": parseFloat($('#position').val())

};

// request method is either POST (create) or PUT (edit)

let methodString, operationString;

if (create) {

methodString = "POST";

operationString = "created";

}

else {

methodString = "PUT";

operationString = "updated";

}

// ajax call

$.ajax({

method: methodString, // either POST or PUT

accepts: 'application/json',

url: URL + "menu", // url is api/menu

contentType: 'application/json', // type of data being sent

data: JSON.stringify(menu), // actual data being sent, use JSON library to convert the menu object to JSON

error: function (jqXHR, textStatus, errorThrown) {

alert('Error: Menu could not be ' + operationString);

},

success: function (result) {

alert('Menu successfully ' + operationString);

var redirectParam = ""

// if this is an edit redirect to that menu display

if (!create) {

redirectParam = "?id=" + menu.id;

}

window.location = "index.html" + redirectParam;

}

});

}

else {

// if there was an error display it

alert(errorString);

}

}

function deleteMenu(id) {

// called by a delete button

// first call changes the button's text to "Confirm Delete?"

// second call sends an ajax request to delete that entry, then gets the menus again

// the confirm delete text

let confirmDeleteText = "Confirm Delete?";

// id of current button

let buttonID = "#deleteMenu" + id;

if ($(buttonID).text() != confirmDeleteText) {

// first call of deleteMenu() changes the text of the calling button

$(buttonID).text(confirmDeleteText)

console.log($(this).text());

}

else {

// second call sends the ajax delete request

// JSON to be sent, only requires the id

let jsonString = '{ "id": ' + id +' }';

$.ajax({

method: "DELETE", // method is DELETE

accepts: 'application/json',

url: URL + "menu", // url is api/menu

contentType: 'application/json', // type of data being sent

data: jsonString, // actual data being sent

success: function (result) {

// Change button text

$(buttonID).text("Successfully Deleted");

// Remove this button's onclick event

$(buttonID).prop('onclick', null).off('click');

// refresh menus

getMenus();

},

error: function (jqXHR, textStatus, errorThrown) {

// Change button text to show delete failure

$(buttonID).text("Delete Failed. There are still sections in this menu");

}

});

}

}

Notice how editMenu() merely redirects to the editMenupage, either with or without an id param.

loadMenu() uses an ajax query to get the menu information and load it into the input fields.

submitMenu() creates a Menu object by pulling the information from the input fields. It then sends either a POST (INSERT) or a PUT (UPDATE) request. The only difference being the POST/PUT and what kind of error message it returns

deleteMenu() runs on the index.html, when the delete button is pressed. The first time it merely changes the button’s text to ask for confirmation. The second time it sends a DELETE request and if successful reloads the menus

**Note**: that submitMenu() has some basic data validation. Using isPositiveNumber(), it checks if position is a valid before submitting the Ajax request. If not, it pops up an error message.

The Menu edit, delete, create buttons, and the editMenu page should now work.

## **10.10 Section: Edit, Delete, Create**

Under wwwroot add a new HTML file item. Name it editSection.html. Replace code with:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Edit Section</title>

<!-- Get the Jquery library from google-->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Load the separate JavaScript file-->

<script src="js/JavascriptFunctions.js"></script>

<script>

// This is the javascript for this page

// global variable denoting whether this is a create or edit

var create = false;

$(document).ready(function () {

// code written here is not executed until the web page has finished loading, preventing any errors with page elements not having loaded yet

// change heading to Create section if that's what we're doing

// a negative id, means this is a create operation, and that negative number is the menuID

if (parseInt(getQueryParam("id")) < 0) {

create = true;

$('#heading').text("Create Section Page");

$('#id').hide();

$('#id').val("0");

$('#idLabel').hide();

// assign the menuID

let menuID = (-1 \* (parseInt(getQueryParam("id"))));

$('#menuID').val(menuID);

console.log("sdgdslg " + menuID);

// change the back link to go to this section's menu

$("#menuDisplayLink").attr("href", "index.html?id=" + menuID);

}

else {

// otherwise load the info for the section that is being edited

loadSection(getQueryParam("id"));

}

// set onclick event for the submit button

$('#submit').click(function () {submitSection(create) });

});

</script>

</head>

<body>

<h2 id="heading"> Edit Section Page</h2>

<div>

<span id="idLabel">ID: </span><br>

<input type="text" id="id" value="" readonly>

<br>

Section Name:<br>

<input type="text" id="name" value="">

<br>

Section Description:<br>

<input type="text" id="description" value="">

<br>

Section Position:<br>

<input type="text" id="position" value="">

<!--MenuID, changeablity not implemented :<br>-->

<input type="hidden" id="menuID" value="">

<!--Picture Path, changeablity not implemented:<br>-->

<input type="hidden" id="picturePath" value="picPath">

<br><br>

<input type="submit" value="Submit" id="submit">

</div>

<br><br>

<a id="menuDisplayLink" href="index.html">Back to Menu Display</a>

</body>

</html>

In JavascriptFunctions.js add these lines:

function editSection(id = null) {

// redirects to the Section Edit/Create page, passes an id as a url parameter if one is passed

let param = "";

if (id != null) {

param = "?id=" + id;

}

window.location = "editSection.html" + param;

}

function loadSection(id) {

// takes a section id and loads it's info into the input elements (id, name, description, and position)

// Jquery ajax call to get the specific section

$.ajax({

method: 'GET',

url: URL + "section/" + id, // The URL we want is api/section/#

dataType: "json",

success: function (section) { // function called if successful

// load data into the input elements

$('#id').val(section.id);

$('#name').val(section.name);

$('#description').val(section.description);

$('#position').val(section.position);

$('#menuID').val(section.menuID);

$('#picturePath').val(section.picturePath);

// change the back link to go to this section's menu

$("#menuDisplayLink").attr("href", "index.html?id=" + section.menuID);

},

error: function (jqXHR, status, errorThrown) { // This function will run if there's an error

alert("ERROR: Can't retrieve that section " + errorThrown + " "); // Pop up a textbox with an error message

}

});

}

function submitSection(create) {

// takes the values of the section input elements (id, name, description, and position)

// and passes them to the api using create if true is passed and using edit if false is passed

// data validation check if position is a number

let errorString = isPositiveNumber($('#position').val(), "Position");

if (errorString == "") {

// create section object by getting value attribute of the input elements

let section = {

"id": parseInt($('#id').val()),

"name": $('#name').val(),

"description": $('#description').val(),

"position": parseFloat($('#position').val()),

"menuID": $('#menuID').val(),

"picturePath": $('#picturePath').val()

};

// request method is either POST (create) or PUT (edit)

let methodString, operationString;

if (create) {

methodString = "POST";

operationString = "created";

}

else {

methodString = "PUT";

operationString = "updated";

}

//console.log(JSON.stringify(section));

// ajax call

$.ajax({

method: methodString, // either POST or PUT

accepts: 'application/json',

url: URL + "section", // url is api/section

contentType: 'application/json', // type of data being sent

data: JSON.stringify(section), // actual data being sent, use JSON library to convert the section object to JSON

error: function (jqXHR, textStatus, errorThrown) {

alert('Error: Section could not be ' + operationString);

},

success: function (result) {

alert('Section successfully ' + operationString);

window.location = "index.html?id=" + section.menuID;

}

});

}

else {

// display error

alert(errorString);

}

}

function deleteSection(id) {

// called by a delete button

// first call changes the button's text to "Confirm Delete?"

// second call sends an ajax request to delete that entry, then gets the sections again

// the confirm delete text

let confirmDeleteText = "Confirm Delete, including all items in this section?";

// id of current button

let buttonID = "#deleteSection" + id;

if ($(buttonID).text() != confirmDeleteText) {

// first call changes the text of the calling button

$(buttonID).text(confirmDeleteText)

}

else {

// second call sends the ajax delete request

// JSON to be sent, only requires the id

let jsonString = '{ "id": ' + id + ' }';

$.ajax({

method: "DELETE", // method is DELETE

accepts: 'application/json',

url: URL + "section", // url is api/section

contentType: 'application/json', // type of data being sent

data: jsonString, // actual data being sent

success: function (result) {

// Change button text

$(buttonID).text("Successfully Deleted");

// Remove this button's onclick event

$(buttonID).prop('onclick', null).off('click');

// refresh current menu

showMenu(currentMenuID);

},

error: function (jqXHR, textStatus, errorThrown) {

// Change button text to show delete failure

$(buttonID).text("Delete Failed");

}

});

}

}

Notice how similar editSection(), loadSection(), submitSection(), and deleteSection() are to the equivalent menu functions.

**Note**: that submitSection() has some basic data validation. Using isPositiveNumber(), it checks if position is a valid before submitting the Ajax request. If not, it pops up an error message.

## **10.11 Item: Edit, Delete, Create**

Under wwwroot, add a new HTML item. Name it editItem.html. Replace code with:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Edit Item</title>

<!-- Get the Jquery library from google-->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Load the separate JavaScript file-->

<script src="js/JavascriptFunctions.js"></script>

<script>

// This is the javascript for this page

// global variable denoting whether this is a create or edit

var create = false;

$(document).ready(function () {

// code written here is not executed until the web page has finished loading, preventing any errors with page elements not having loaded yet

// set menuID that we should return to after a sucessful create/edit

if (parseInt(getQueryParam("menuID")) > 0) {

var menuID = parseInt(getQueryParam("menuID"));

// change the back link to go to this item's menu

$("#menuDisplayLink").attr("href", "index.html?id=" + menuID);

}

else {

var menuID = 1;

}

// change heading to Create item if that's what we're doing

// a negative id, means this is a create operation, and that negative number is the sectionID

if (parseInt(getQueryParam("id")) < 0) {

create = true;

$('#heading').text("Create Item Page");

$('#id').hide();

$('#id').val("0");

$('#idLabel').hide();

// assign the sectionID

$('#sectionID').val(-1 \* (parseInt(getQueryParam("id"))));

}

else {

// otherwise load the info for the item that is being edited

loadItem(getQueryParam("id"));

}

// set onclick event for the submit button

$('#submit').click(function () {submitItem(create, menuID) });

});

</script>

</head>

<body>

<h2 id="heading"> Edit Item Page</h2>

<div>

<span id="idLabel">ID: </span><br>

<input type="text" id="id" value="" readonly>

<br>

Item Name:<br>

<input type="text" id="name" value="">

<br>

Item Description:<br>

<input type="text" id="description" value="">

<br>

Item Position:<br>

<input type="text" id="position" value="">

<!--sectionID, changeablity not implemented :<br>-->

<input type="hidden" id="sectionID" value="">

<!--Picture Path, changeablity not implemented:<br>-->

<input type="hidden" id="picturePath" value="picPath">

<br>

<input type="submit" value="Submit item changes" id="submit">

<br />

<br />

</div>

<div id="pricelineDiv">

</div>

<br><br>

<a id="menuDisplayLink" href="index.html">Back to Menu Display</a>

</body>

</html>

In JavascriptFunctions.js add the following:

function editItem(itemID, menuID) {

// redirects to the Item Edit/Create page,

// first parameter is either a positive item or a negative section id

// (a negative number signifies that this is a create operation)

// second parameter is a menuID

let param = "?id=" + itemID + "&menuID=" + menuID;

window.location = "editItem.html" + param;

}

function loadItem(id) {

// takes an item id and loads it's info into the input elements (id, name, description, position, etc)

// Also adds elements describing its associated pricelines to the pricelineDiv

// Jquery ajax call to get the specific item

$.ajax({

method: 'GET',

url: URL + "item/" + id, // The URL we want is api/item/#

dataType: "json",

success: function (item) { // function called if successful

$('#id').val(item.id);

$('#name').val(item.name);

$('#description').val(item.description);

$('#position').val(item.position);

$('#sectionID').val(item.sectionID);

$('#picturePath').val(item.picturePath);

// empty elements from the PRICELINE\_DIV\_ID

$(PRICELINE\_DIV\_ID).empty();

// add a button to add another priceline form

$('<button class="edit" id="newPricelineButton"> Add new priceline </button> <br/> <br/> ').appendTo($(PRICELINE\_DIV\_ID));

// add onclick event to add the form and hide this button

$('#newPricelineButton').click(function() {

listPriceline(null, item.id);

$(this).hide();

});

// loop through pricelines

for (let i = 0; i < item.priceLineList.length; i++) {

let priceline = item.priceLineList[i];

listPriceline(priceline);

}

},

error: function (jqXHR, status, errorThrown) { // This function will run if there's an error

alert("ERROR: Can't retrieve this item " + errorThrown + " "); // Pop up a textbox with an error message

}

});

}

function listPriceline(priceline, itemID) {

// takes a priceline object and adds input fields with it's contents to the PRICELINE\_DIV\_ID

// and also adds submit changes and delete priceline buttons

// if the priceline object is null, it substitutes an empty priceline object with id = 0 (for a create operation, where id doesn't matter)

// the second parameter is an itemID, given when this is a create

//console.log(priceline);

// if it's null then this is a create

if (priceline == null) {

var create = true;

priceline = {

"id": 0,

"description": "",

"price": "",

"position": "",

"itemID": itemID

};

console.log(priceline);

}

else {

var create = false;

}

// create the html string of input elements

var htmlString = "";

if (create) {

htmlString += '<span>New Priceline Submission: </span><br>';

}

htmlString +=

'<span>Priceline Description </span><br>' +

'<input id="pricelineDescription' + priceline.id + '" type = "text" value = "' + priceline.description + '"> <br />' +

'<span>Priceline Price ($) </span><br>' +

'<input id="pricelinePrice' + priceline.id + '" type = "text" value = "' + priceline.price + '"> <br />' +

'<span>Priceline Position </span><br>' +

'<input id="pricelinePosition' + priceline.id + '" type = "text" value = "' + priceline.position + '">' +

'<input id="pricelineItemID' + priceline.id + '" type = "hidden" value = "' + priceline.itemID + '">' +

'<input id="pricelineID' + priceline.id + '" type = "hidden" value = "' + priceline.id + '"> <br />';

// only want a submit button if this is a create

if (create) {

htmlString += '<button class="edit" onclick="submitPriceline(' + priceline.id + ', true)"> Submit New Priceline </button>' // submit button;

}

else {

htmlString +=

'<button class="edit" onclick="submitPriceline(' + priceline.id + ', false)"> Submit Priceline Changes</button>' + // submit changes button

'<button id="deletePriceline' + priceline.id + '" class="delete" onclick="deletePriceline(' + priceline.id + ', ' + priceline.itemID + ')"> Delete this Priceline</button> <br /> <br /> ' // delete priceline button

}

htmlString += '<br /> ';

// if this is a create add to beginning of div

if (create) {

$(htmlString).prependTo($(PRICELINE\_DIV\_ID));

}

else {

// otherwise add to end of div

$(htmlString).appendTo($(PRICELINE\_DIV\_ID));

}

}

function submitItem(create, menuID = 1) {

// takes the values of the item input elements (id, name, description, and position)

// and passes them to the api using create if true is passed and using edit if false is passed

// second parameter is the menuID, used for the direct back to the display menu page (defaults to 1)

// datavalidation check if position is a number

let errorString = isPositiveNumber($('#position').val(), "Item Position");

if (errorString == "") {

// create item object by getting value attribute of the input elements

let item = {

"id": parseInt($('#id').val()),

"name": $('#name').val(),

"description": $('#description').val(),

"position": parseFloat($('#position').val()),

"sectionID": $('#sectionID').val(),

"picturePath": $('#picturePath').val()

};

// request method is either POST (create) or PUT (edit)

let methodString, operationString;

if (create) {

methodString = "POST";

operationString = "created";

}

else {

methodString = "PUT";

operationString = "updated";

}

console.log(JSON.stringify(item));

// ajax call

$.ajax({

method: methodString, // either POST or PUT

accepts: 'application/json',

url: URL + "item", // url is api/item

contentType: 'application/json', // type of data being sent

data: JSON.stringify(item), // actual data being sent, use JSON library to convert the item object to JSON

error: function (jqXHR, textStatus, errorThrown) {

alert('Error: Item could not be ' + operationString);

},

success: function (result) {

alert('Item successfully ' + operationString);

window.location = "index.html?id=" + menuID;

}

});

}

else {

// display errorString

alert(errorString);

}

}

function deleteItem(id) {

// called by a delete button

// first call changes the button's text to "Confirm Delete?"

// second call sends an ajax request to delete that entry, then gets the items again

// the confirm delete text

let confirmDeleteText = "Confirm Delete, including all this item's prices?";

// id of current button

let buttonID = "#deleteItem" + id;

if ($(buttonID).text() != confirmDeleteText) {

// first call changes the text of the calling button

$(buttonID).text(confirmDeleteText)

console.log($(this).text());

}

else {

// second call sends the ajax delete request

// JSON to be sent, only requires the id

let jsonString = '{ "id": ' + id + ' }';

$.ajax({

method: "DELETE", // method is DELETE

accepts: 'application/json',

url: URL + "item", // url is api/item

contentType: 'application/json', // type of data being sent

data: jsonString, // actual data being sent

success: function (result) {

// Change button text

$(buttonID).text("Successfully Deleted");

// Remove this button's onclick event

$(buttonID).prop('onclick', null).off('click');

// refresh current menu

showMenu(currentMenuID);

},

error: function (jqXHR, textStatus, errorThrown) {

// Change button text to show delete failure

$(buttonID).text("Delete Failed");

}

});

}

}

Notice that for the most part, editItem(), loadItem(), submitItem(), and deleteItem() are very similar to the equivalent menu and section functions.

The main difference is that the editItem.html page also lists the pricelines. This is done by having loadItem() call listPriceline() for each element in the item’s pricelineList.

listPriceline() takes a priceline object and creates the necessary <p> elements describing it and also edit and delete buttons. If it was given null instead of a priceline object and then this is a create and it leaves the input elements blank and has a submit button instead.

**Note**: that submitItem() has some basic data validation. Using isPositiveNumber(), it checks if position is a valid before submitting the Ajax request. If not, it pops up an error message.

## **10.12 Priceline: Edit, Delete, Create**

There is no editPriceline.html page. Edits, deletes, and creates for priceline are handled on the editItem.html page.

In JavascriptFunctions.js add these final lines:

function submitPriceline(pricelineID, create) {

// takes a pricelineID and then grabs the values from the input elements (id, description, price, and position)

// and passes them to the api using create if true is passed and using edit if false is passed

// data validation: check if position and price are numbers

let errorString = isPositiveNumber($('#pricelinePrice' + pricelineID).val(), "Priceline Price") + isPositiveNumber($('#pricelinePosition' + pricelineID).val(), "Priceline Position");

if (errorString == "") {

// create priceline object by getting value attribute of the input elements

let priceline = {

"id": parseInt($('#pricelineID' + pricelineID).val()),

"description": $('#pricelineDescription' + pricelineID).val(),

"price": parseFloat($('#pricelinePrice' + pricelineID).val()),

"position": parseFloat($('#pricelinePosition' + pricelineID).val()),

"itemID": $('#pricelineItemID' + pricelineID).val()

};

// request method is either POST (create) or PUT (edit)

let methodString, operationString;

if (create) {

methodString = "POST";

operationString = "created";

}

else {

methodString = "PUT";

operationString = "updated";

}

console.log(JSON.stringify(priceline));

// ajax call

$.ajax({

method: methodString, // either POST or PUT

accepts: 'application/json',

url: URL + "priceline", // url is api/priceline

contentType: 'application/json', // type of data being sent

data: JSON.stringify(priceline), // actual data being sent, use JSON library to convert the priceline object to JSON

error: function (jqXHR, textStatus, errorThrown) {

alert('Error: Priceline could not be ' + operationString);

},

success: function (result) {

alert('Priceline successfully ' + operationString);

// reload this item and its pricelines

loadItem(priceline.itemID);

}

});

}

else {

// display errorString

alert(errorString);

}

}

function deletePriceline(id, itemID) {

// called by a delete button

// first call changes the button's text to "Confirm Delete?"

// second call sends an ajax request to delete that entry, then refreshes the item listing (id given by 2nd parameter)

// the confirm delete text

let confirmDeleteText = "Confirm Delete?";

// id of current button

let buttonID = "#deletePriceline" + id;

if ($(buttonID).text() != confirmDeleteText) {

// first call changes the text of the calling button

$(buttonID).text(confirmDeleteText)

console.log($(this).text());

}

else {

// second call sends the ajax delete request

// JSON to be sent, only requires the id

let jsonString = '{ "id": ' + id + ' }';

$.ajax({

method: "DELETE", // method is DELETE

accepts: 'application/json',

url: URL + "priceline", // url is api/priceline

contentType: 'application/json', // type of data being sent

data: jsonString, // actual data being sent

success: function (result) {

// Change button text

$(buttonID).text("Successfully Deleted");

// Remove this button's onclick event

$(buttonID).prop('onclick', null).off('click');

// reload the item and its pricelines

loadItem(itemID);

},

error: function (jqXHR, textStatus, errorThrown) {

// Change button text to show delete failure

$(buttonID).text("Delete Failed");

}

});

}

}

There are no editPriceline() or loadPriceline() functions. The loadPriceline() functionality is supplied by listPriceline() from the previous section.

The submitPriceline() and deletePriceline() functions are quite similar to the section and item equivalents.

**Note**: that submitPriceline () has some basic data validation. Using isPositiveNumber(), it checks if position **AND** price are valid numbers before submitting the Ajax request. If not, it pops up an error message.

## **11 Issues and Resolutions**

There might by any number of typos inside this guide, for which I apologize. There shouldn’t be any within the coding sections, but if there are you can always copy the relevant code from the KiabaReduxAPI GitHub repository. That code repository works on multiple Windows machines, so it should work as long as you follow the installation instructions given in Section 3.

This is a large and complex project. If you’re having trouble understanding part of it, we encourage you to ask someone else if they understand it. Coding is a collaborative activity. Always seek your peers’ knowledge and ideas. If you can’t seek the advice of people physically around you, then remember that there is a wealth of knowledge available online.

If you don’t understand or know something, Google (and the internet in general) should probably be your first response. Other useful sites include:

<https://stackoverflow.com/>

Where both novices and experts ask all kinds of questions from the basic to the very obscure.

<https://www.w3schools.com/>

I’ve listed W3 Schools pages throughout this guide, because I honestly believe them to be the best free coding resource on the web. They have lessons on HTML, JavaScript, CSS, PHP, Python, and more.

<https://www.codecademy.com/catalog/subject/all>

Codecademy is another great resource. They do charge for their premium material, but as a free member, you can still do a substantial part of their courses.

## **12 Improvement Suggestions**

This project is far from perfect. Here are a few improvements that could be made.

**12.1 Parameterized SQL Queries**

A fairly simple change would be to replace all the current dynamic SQL queries with parameterized ones (aka prepared statements).

This will both prevent SQL injection and help to keep your database’s data consistency uniform. SQL injection is a security vulnerability that can allow a malicious hacker to do anything they want to your database.

For more info see: <https://www.w3schools.com/sql/sql_injection.asp>

This application would be better secured using parameterized statements. These are not only generally faster, but also much more secure and make SQL injection almost impossible. For more info on prepared statements see:

<https://csharp-station.com/Tutorial/AdoDotNet/Lesson06>

**12.2 Actual Login Authentication**

Currently, the project just allows you to login and see the edit and delete buttons without an identity check of any sort. Add a username and password box. Have the login button send an ajax request to the API to check if that username/password combo is valid. Only then are the edit and delete buttons shown. (the usernames/passwords could be stored in the code or for bonus points stored in the database).

You might consider using ASP.NET’s Identity framework. See: <https://docs.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity>

**12.3 Web Application Security**

If you’ve taken CIS 491, then you should be familiar with some of the concepts of web app security. They include Cross-Site Scripting (CSS) protection, Cross-Site Request Forgery (CSRF) protection, Secure User authentication (storing usernames and salted, encrypted passwords in a database to use for authentication, and Server-side data validation (a truly secure and resilient application checks data both on the client-side, like this one does, and then again on the server side)

See this Linda.com course for more info on these topics. Should be free by logging in with your Bronco account.

<https://www.lynda.com/Web-Development-tutorials/Foundations-Programming-Web-Security/133330-2.html>

**12.4 Add Some Pretty Pictures**

One of the original goals of this project was to have pictures for each section and item. And allow the pictures to be changed and even uploaded by an authorized user. This had to be dropped due to time constraints, but the bare bones of it still exists within the database and DbAccessManagement class.

In order to do this, you would need to

* Create an images folder in wwwroot
  + Populate it with images
* Assign each section and item in the database a picPath that corresponds to a picture
  + Ex. picPath = “onion rings.jpg”
* Add <img> elements for each section and item have them point to an image
  + src="images/" + picPath

To add change picture functionality, you’d need to either add a picPath <input> element to the editItem and editSection HTML pages or create a selectPicture.html page that shows all available pictures and uses radio buttons to allow the user to select one

For picture uploads, that’s a super challenge. You’re just gonna have to Google file uploads.

**12.5 Online Ordering**

Add a cart and the ability to place online orders through the website.

One way to do this might be to have a separate onlineOrdering.html page with checkboxes and/or comboboxes to select items and quantity. Then have that page send some JSON back to the API, which sends back an order confirmation of some kind. (For bonus points maybe it stores the order in the database and for super bonus points it can list out your previous orders)

**12.6 Be Creative**

As it is this project is pretty bland. There’s no color, no fonts, and no formatting. Maybe you could add some CSS to improve this website’s look. An excellent library to try is Bootstrap, the most popular web framework for creating responsive web pages that look great on both on mobile devices and larger laptop screens. Bootstrap has many functionalities already built in like equally spaced columns and responsive HTML elements (elements that dynamically resize based on the page size, so things look good on different sized mobile screens).

See: <https://www.w3schools.com/bootstrap4/default.asp>

**12.7 Be Creative (In a More Metaphorical and Less Artistic Sense)**

There’s plenty other things to do with this project. Have fun, go crazy. Set some crazy goal and strive to reach it. Even if you fall short, you’ll have learned by doing something crazy, which is really what this project should be about.