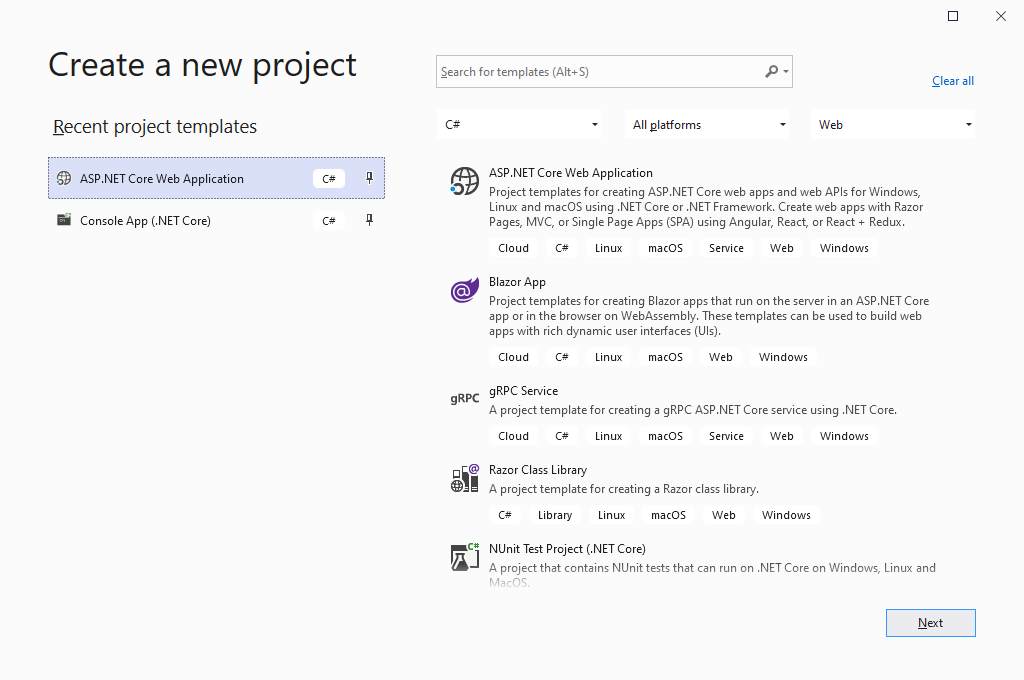
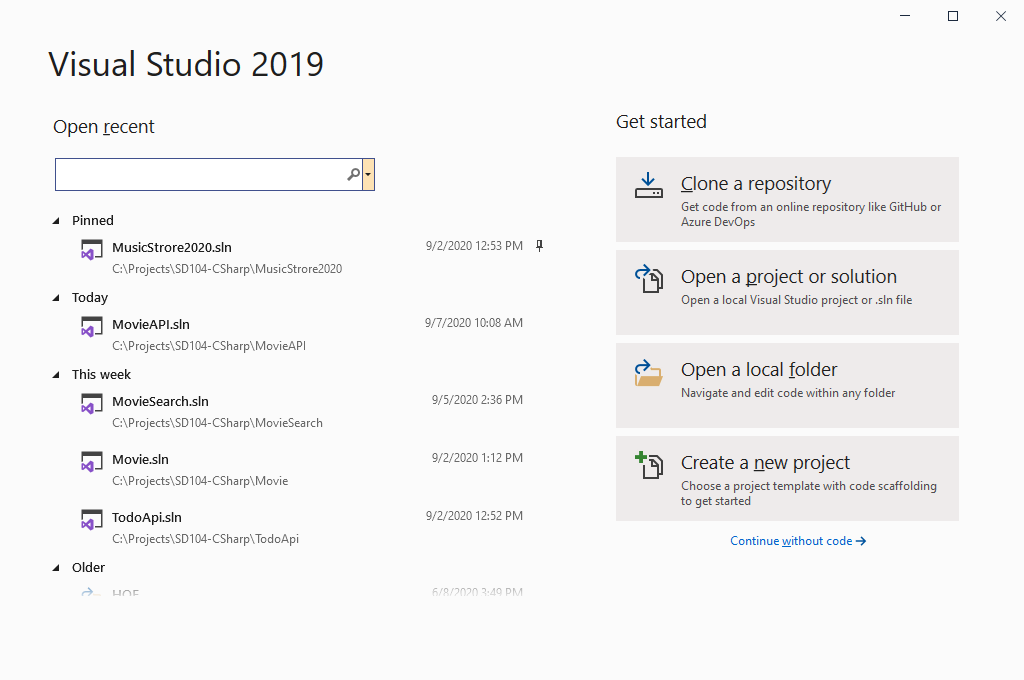
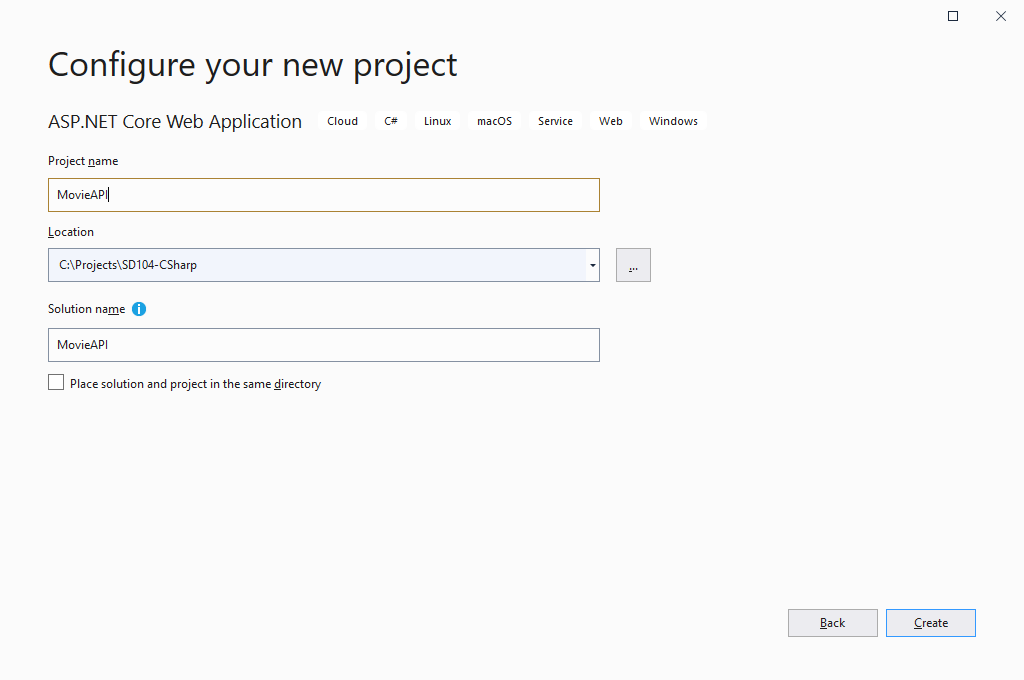
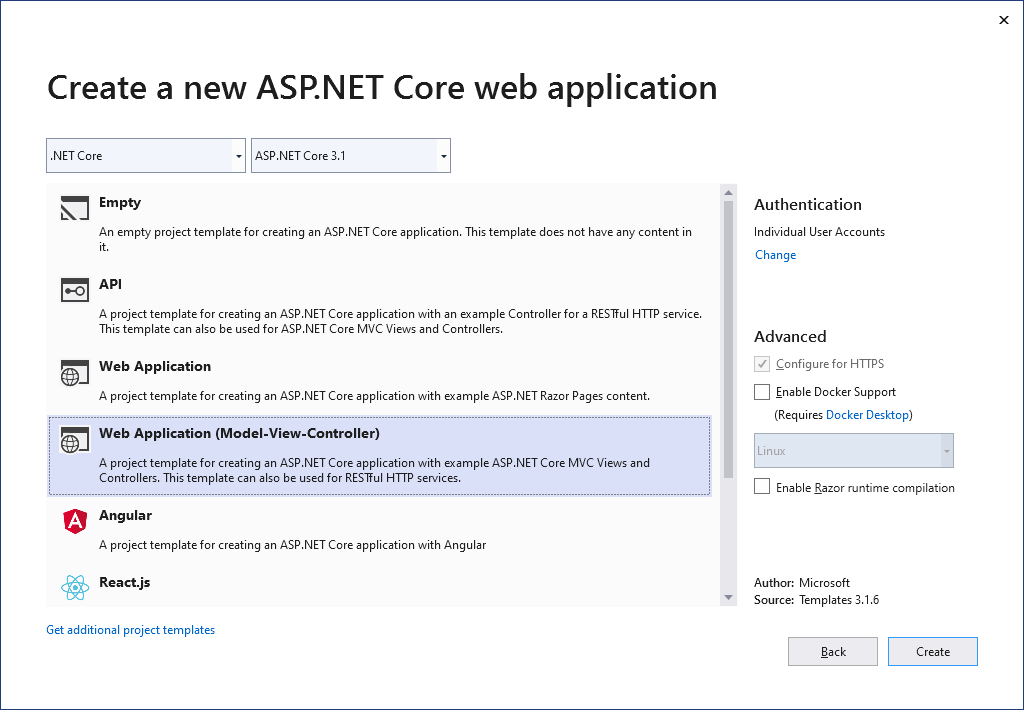
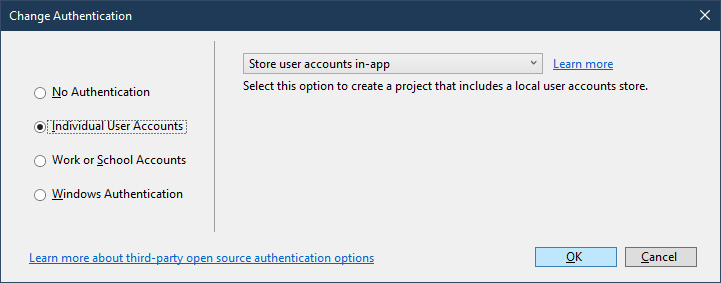
# Create a Web application







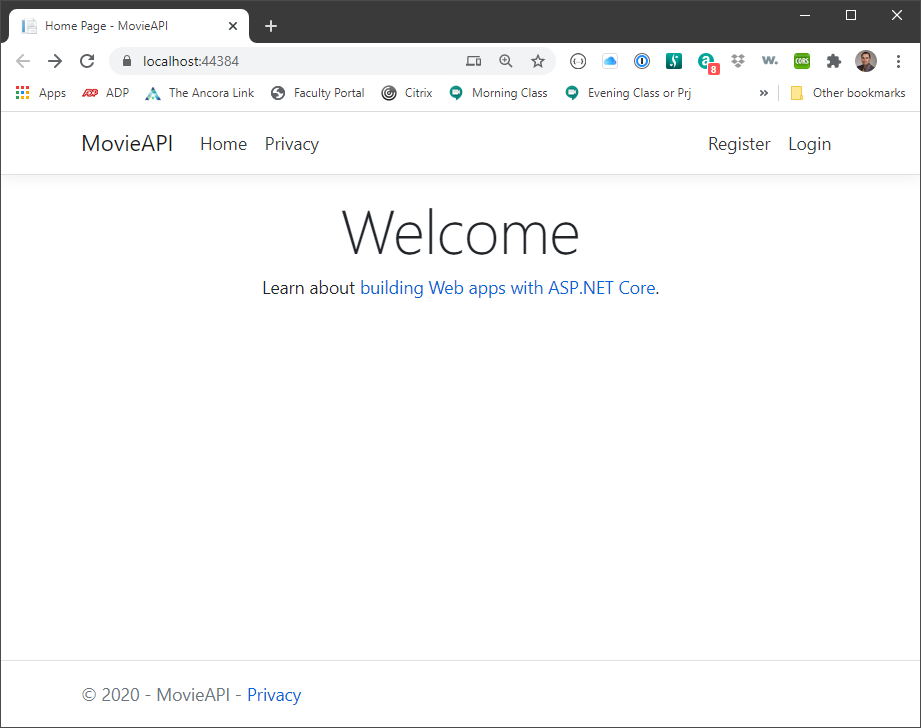
Click on the Individual User Accounts Change link. This will create our web page with the ability to register users on our site.

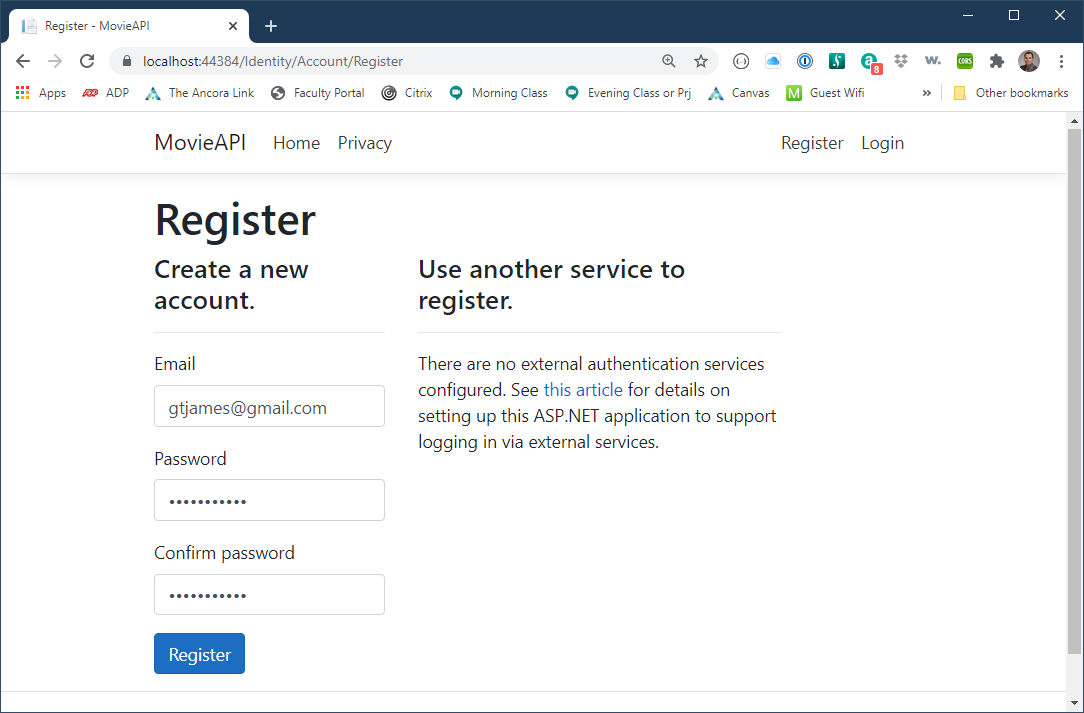


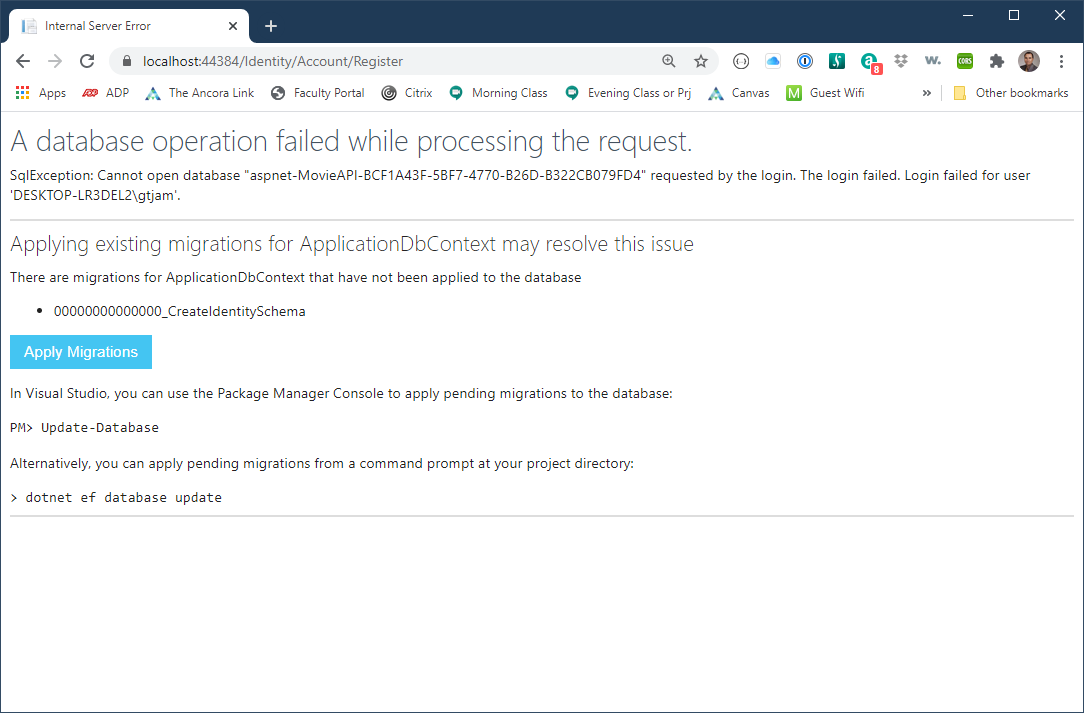
Visual Studio has now created a very simple Web App with user authentication.

# Take it for a spin

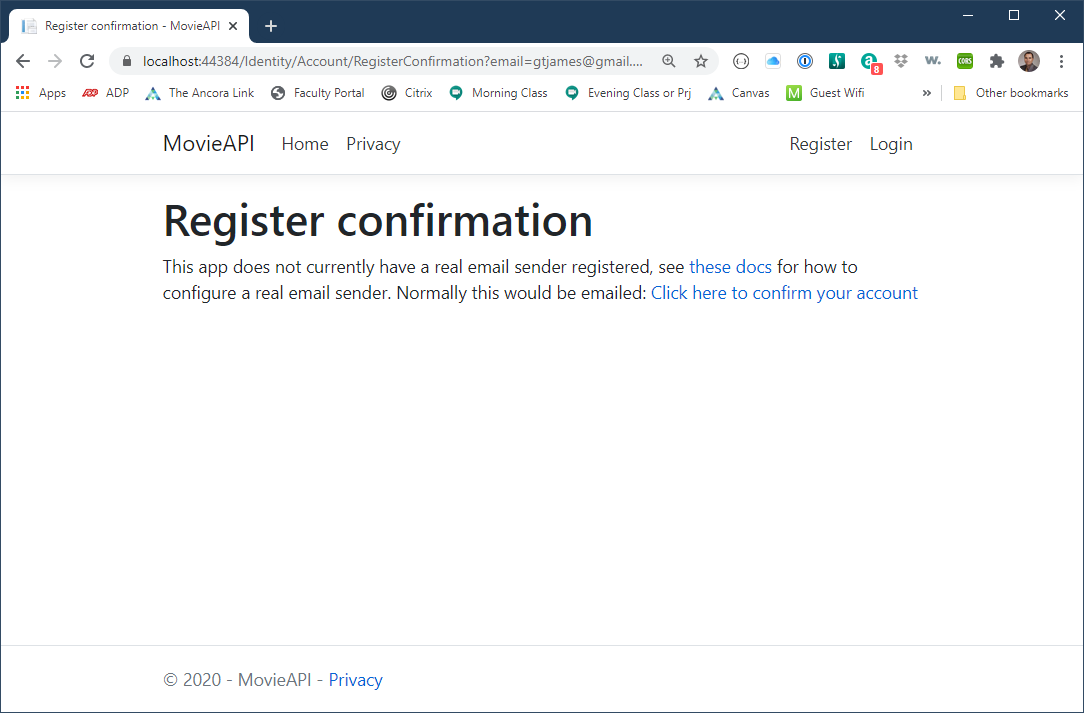
Click the run button. When it opens click on the register link.



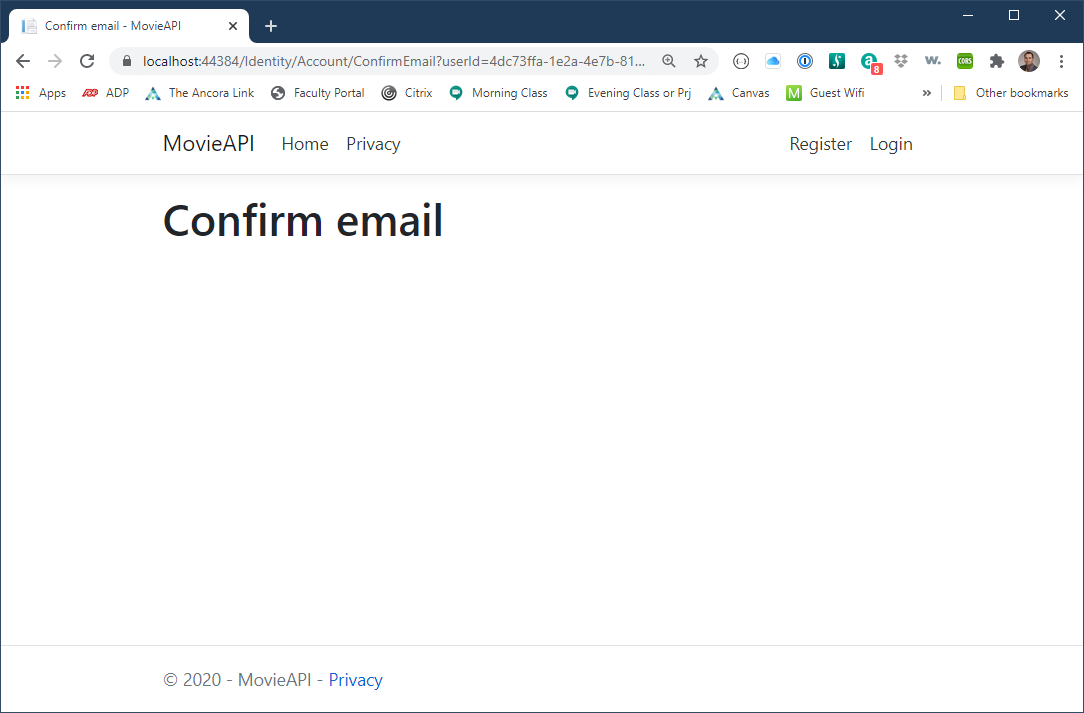




That’s right we didn’t do the Add-Migration command in the Package Manager Console. Click on the Apply Migration button, wait 10 seconds or so and refresh the page. Your user has been created!



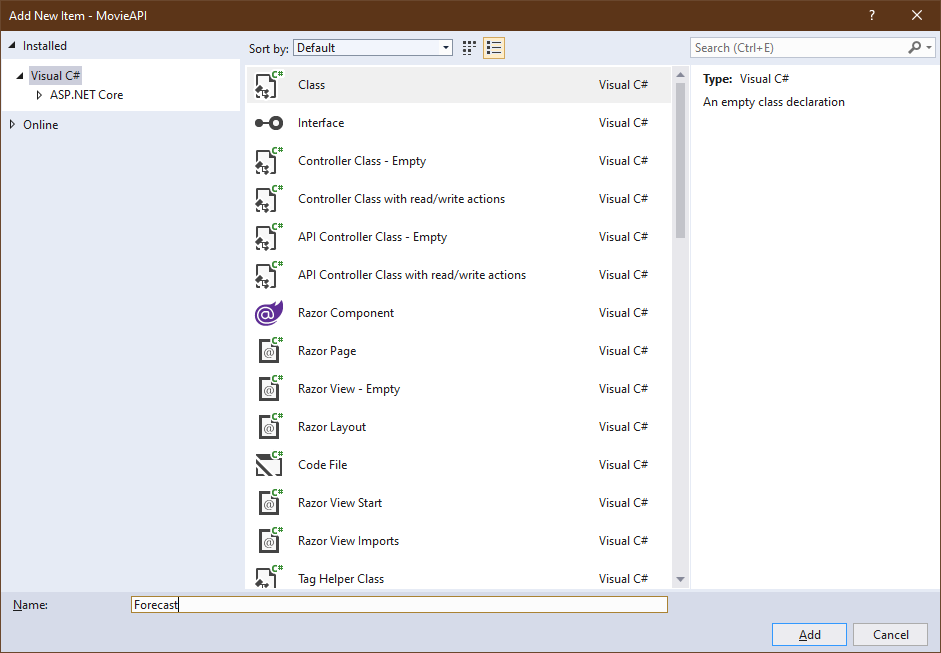
Click the Click here to confirm your account button.



Boom! You are registered. Close the app and return to VS.

# Add our First API data Object - Forecast

Right click on Models. Create new Class **Forecast**



Add these attributes to the class

public DateTime Date { get; set; }

public int TemperatureC { get; set; }

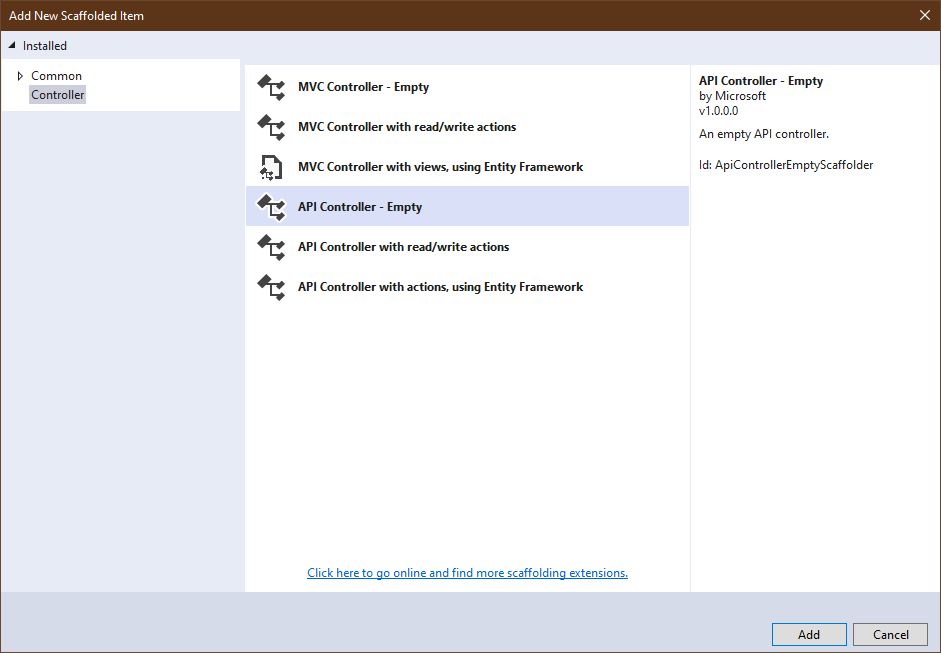
public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

public string Summary { get; set; }

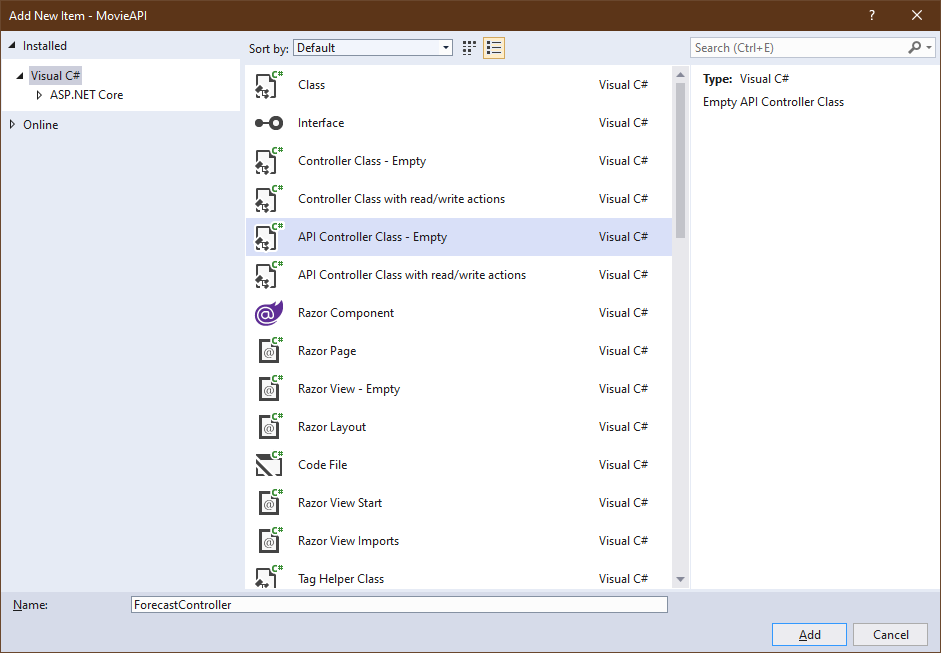
We have added in a small Lambda expression to convert Celsius to Fahrenheit.

# Add API Controller – **Empty**

Right click on the **Controllers** folder. Click on **Add** and then **Controllers**. You can add an API Controller with actions, using Entity Framework but we are going to build ours from scratch. Select the **API Controller – Empty option**



Name it ForecastController



# Add an endpoint for the Forecast API

Insert this code into your ForecastController class

// GET: api/<Forecast>

/\*

\* Just a simple API. Browswer or Postman does a api/forecast GET

\* a 5 day JSON forecast is returned

\*

\*/

[HttpGet]

public IEnumerable<Forecast> Get()

{

string[] Summaries = { "Freezing", "Bracing", "Chilly", "Cool", "Mild",

"Warm", "Balmy", "Hot", "Sweltering", "Scorching" };

var rng = new Random(); // create a random number generator

// a bit of fancy C#.

// return returns an array of Forecast objects

// Enumerable.Range(1, 5) create an array of numbers 1 to 5

// OK! it is not an array it is an Enumerable list

// Select(index => take each number and apply to the next LOC

// new Forecast create a new Forecast object

// .ToArray(); take each Forecast object and add to an array

return Enumerable.Range(1, 5).Select(index => new Forecast

{ // create a date + index days into the future

Date = DateTime.Now.AddDays(index),

TemperatureC = rng.Next(-20, 55), // random temp between -20 and 55C

// random weather condition

Summary = Summaries[rng.Next(Summaries.Length)]

})

.ToArray();

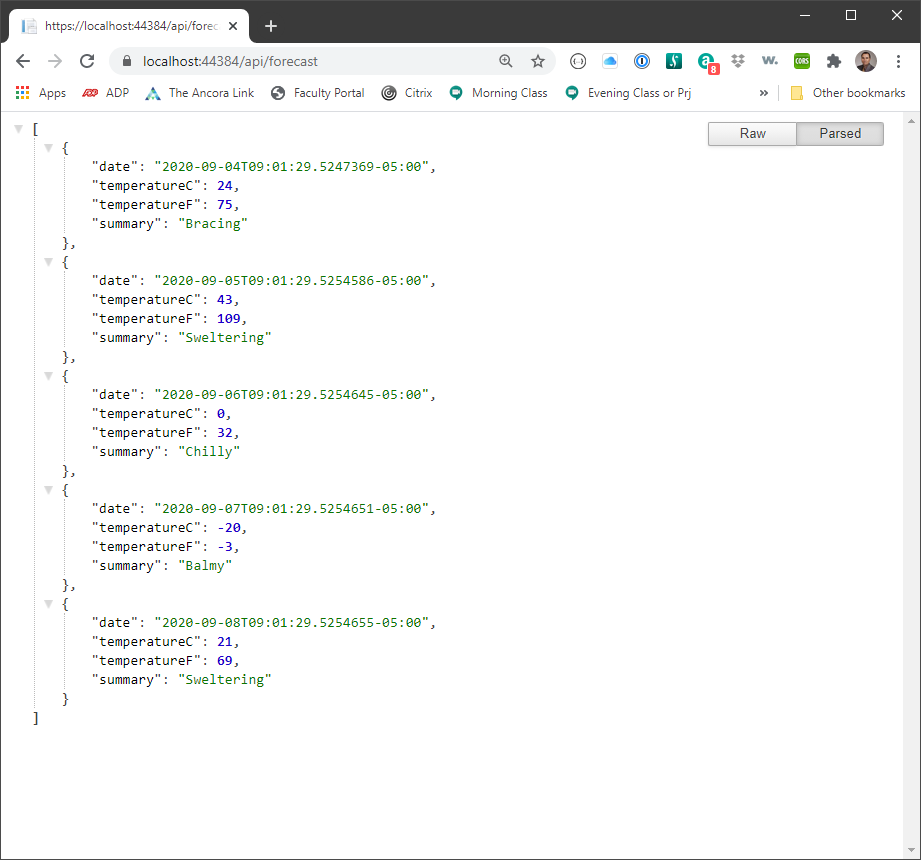
}

The Forecase object reference will be new to this class so it will generate an error but the editor will help you by asking if you would like the Models package added to your code. This will add a new using line of code at the top of the file.

# Testing our first API

Run the application. The home page will appear. Add /api/forecast to the URL, press enter.

Our API will be requested and BOOM! We get our data



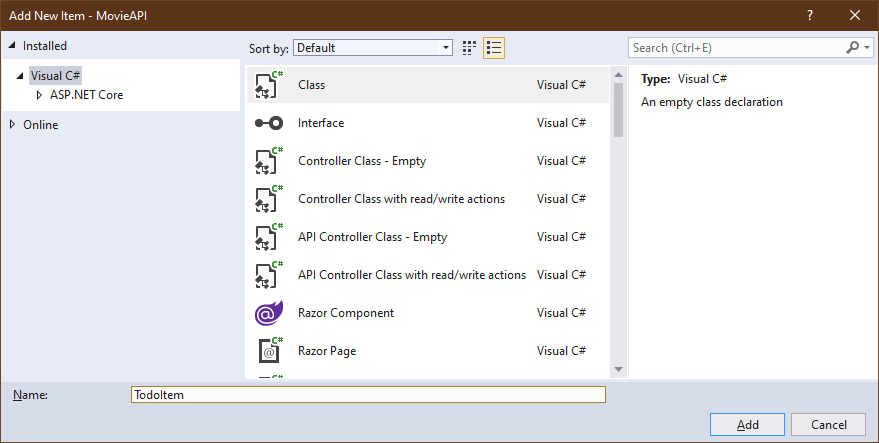
The code you just added to the API Controller created an array of Forecast objects. Here you see the results of that code.

Close the app and return to VS

# Create next POCO – TodoItem

Next on our agenda is to create new data entity and the APIs to manage it. This will be a simple To-Do item

Create a new class in the Models folder. Call it TodoItem



Add these attributes

public long Id { get; set; }

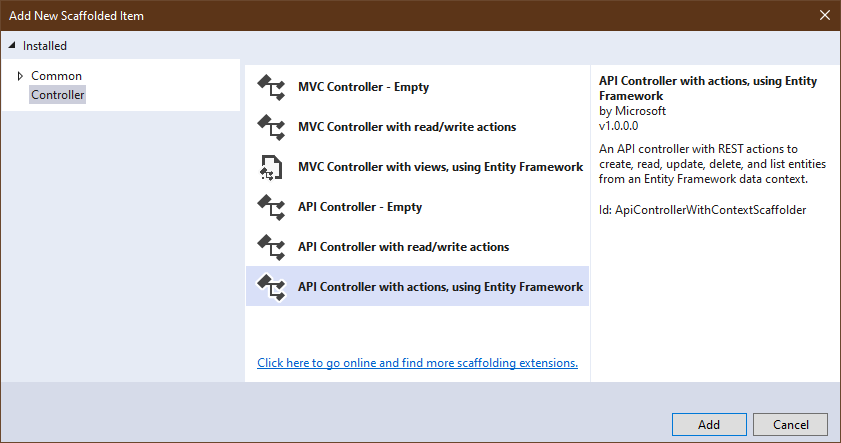
public string Name { get; set; }

public bool IsComplete { get; set; }

Add these attributes INSIDE the class. We will be added this class to our database. Which is a big reason why we need the Id field. It will be the unique identifier (key) to our record in the database.

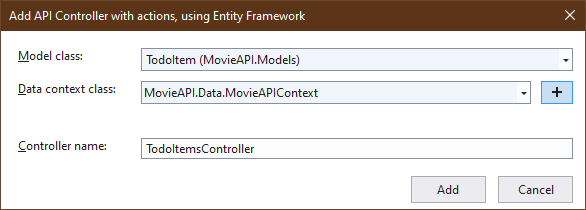
# Create the next API Controller – TodoItemsController

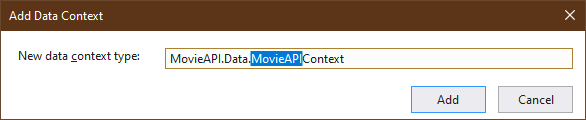
Create a new controller. Our next controller will be **API Controller with actions, using Entity Framework.**



**Model class**: Select the POCO we just created. TodoItem

**Data context class**: We will create our own. Click the plus button and accept the default data context type suggested in the **Add Data Context** window.





The constructed Controller will have all of the APIs to perform CRUD operations using the HTTP verbs (POST-Create, GET-Read, PUT-Update, and DELETE-Delete)

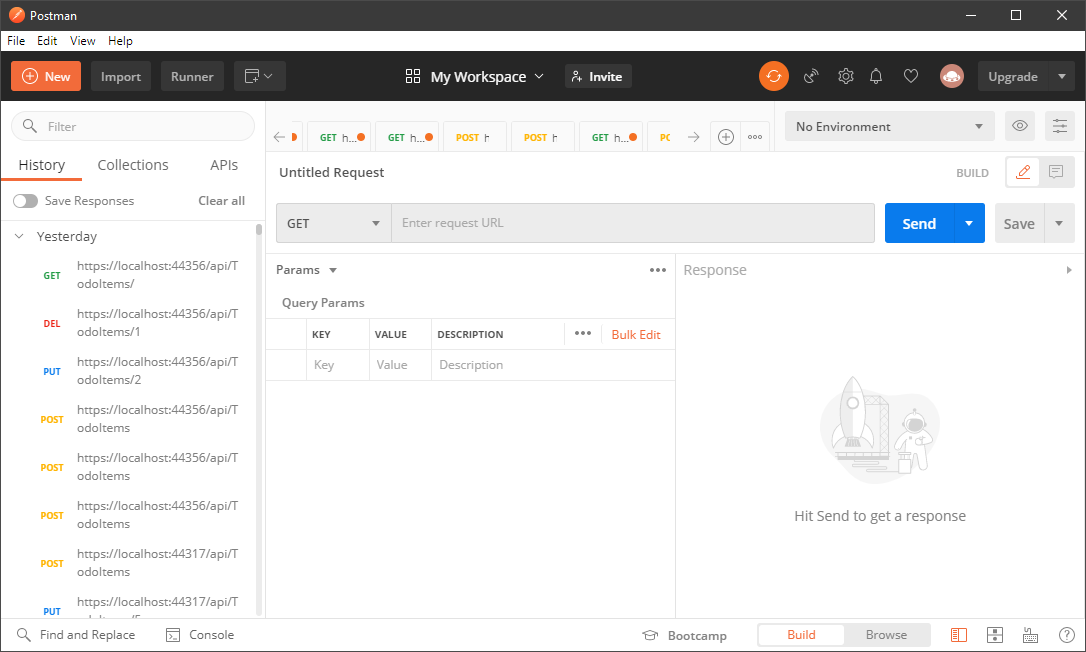
Before we test, we need to migrate our POCO to the database. We need to specify which database context to use. The application has its own (invisible) context for managing the user. We just created our own application database context for the records we are creating to go into our database. So as we add new POCOs we which to manage we need to migrate them to the database use the context we have created.

add-migration -Context **MovieAPIContext** todoItems

update-database –Context **MovieAPIContext**

Anything besides GET will have to be tested using Postman. Run your application. The Home page will open. Copy the URL from the address bar. We are going to need the port number for the Postman requests. We will test the GET request once in the browser. With the home page open, add /api/TodoItems to the address URL and press return. And we get back an empty array or objects []. Well it is an empty database so that is to be expected. How do we add records without a CRUD application with all of the nice MVC endpoints? We will use Postman.

# Open Postman



We will test the list all TodoItems first. Make sure that GET is selected for the HTTP action. Enter in the URL from the browser. Add /api/todoitems to the URL. Press return and Baam! Nothing too exciting. Just []; just like we got in the browser test.

# Create a TodoItem using Postman

Change the HTTP verb from GET to POST

Leave the URL alone

In the window below the Request fields there are several tabs. Params – Auth – Headers – **Body** – Pre-req – Tests – Settings. Click on the **Body** tab

This will create a text area to add our JSON text to POST to our API. Paste the following into the text area.

{

  "name":"mow lawn",

  "isComplete":false

}

We are ready to go. Press the Send button

In the right hand panel, you will see the results of your POST

{

    "id": 1,

    "name": "mow lawn",

    "isComplete": false

}

What just happened? Our JSON data was passed to our PostTodoItem method in our TodoItemsController.

public async Task<ActionResult<TodoItem>> PostTodoItem**(TodoItem todoItem)**

{

\_context.TodoItem.**Add**(todoItem);

await \_context.**SaveChangesAsync**();

return **CreatedAtAction**("GetTodoItem", new { id = todoItem.Id }, **todoItem**);

}

The code is around line 81. The JSON data resembles a TodoItem. So, .NET will convert the JSON text into a **TodoItem** which we will **Add** to the database. The last line of the method will trigger the GetTodoItem action (see line 33); whose purpose is to return a **todoItem** object to the requester (Postman).

Let’s rerun the GET all TodoItems request. Look at the left hand History panel in Postman. There is a growing list of all of your requests.

Find the most recent GET. Click on it, the Request fields will be populated and then press Send. Now you see the JSON data for the item we just added.

Click on your POST item and change some of the data. Press Send. Do this several times to add a handful of items to your database.

Once more, do the GET request and see the results of your code. Return to your browser and refresh the page. You will see your data return in JSON format.

Add a **/3** to the end of your URL request. Your results will show you the details of a single TodoItem with the id of 3. We just exercised a new endpoint.

# Updating data using the PUT request

Change the GET verb now to PUT. Your URL should still look something like this

https://localhost:44384/api/todoItems/3

Your port number will be different.

To do the update we need to update the data, including adding the id of the entity we wish to update.

{

"id":3,

"name":"walk dog again",

"isComplete":false

}

Press Send

Select the GET request if it has the /3 at the end go ahead and use that one. Press Send and it will return the updated value of this record. Remove the /3 and press Send and all records will be returned again. Notice that record 3 has been changed.

If you want to delete a record use the same URL with the /3. Change the HTTP verb to DELETE. Press Send. Then do the GET for the select all. Your record 3 has been deleted.

# Get Registered with IMDB-API

Next, we are going to set up a third-party API request to IMDB. Register with IMDB to get your own key. It is free.

<https://imdb-api.com/Identity/Account/Register>

It takes 2 minutes to register and to get your confirmation email

Login to IMDB-API. Click on the API tab and **save your API Key**

There are many API requests we can do with IMDB. We are just going to do a very simple one. We will request a list of movies that match a title we will provide.

We can do that right now. Enter this URL into the address bar of your browser. Use YOUR key not the k\_XXXXX you see here.

<https://imdb-api.com/en/API/SearchMovie/k_XXXXXXXX/casablanca>

Look at the results. You see header data about the search as well as the result attribute, which is an array of details about the movie title you entered.

# Setting up the Movie Entities

The next POCOs we will build will reflect what is returned from IMDB.

MovieHeader and Movie

Create the Movie class in the Models folder

Add these attributes

public string id { get; set; }

[Display(Name = "IMDB Result")]

public string resultType { get; set; }

public string image { get; set; }

public string title { get; set; }

public string description { get; set; }

I tossed in a Display annotation for good measure. You will need to click on the Yellow lightbulb to the left of that line of code to add the appropriate using line to the top of the class

using System.ComponentModel.DataAnnotations;

Next, create the MovieHeader class in the Models folder

public string searchType { get; set; }

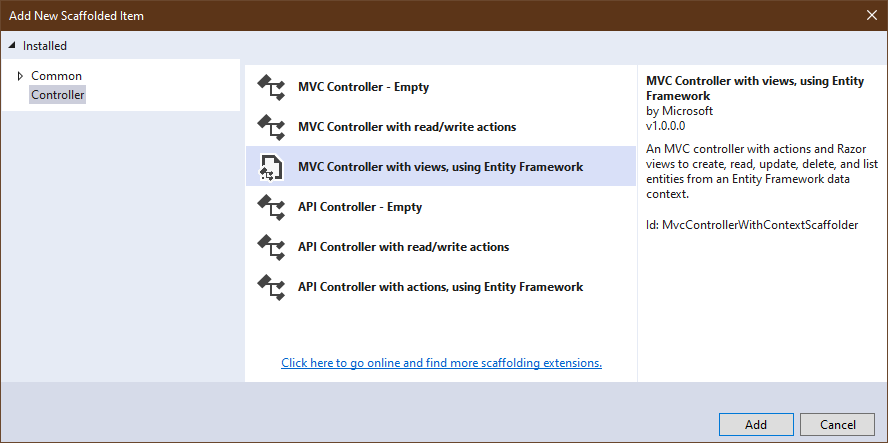
public string expression { get; set; }

public Movie[] results { get; set; }

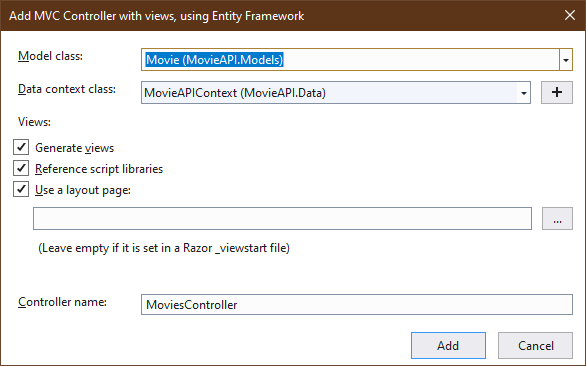
public string errorMessage { get; set; }

# Create the MoviesController

We will build an **MVC Controller with views using Entity Framework.** We will create two ways to get Movie data from IMDB. One use the MVC Controller with web pages to read and update the database. Our second method we will use API request that could be used in an AJAX call from JavaScript code in a web page.



Select the Movie class for the Model class



Pressing Add will generate the MoviesController and the .cshtml files for the MVC

# Add the IMDB API request

Paste the code below to the **MoviesController.cs** code. You will paste it just after the closing curly brace for the **MovieExists** method. It will leave your code with a number of errors because of the new packages that are required. Click on the Yellow lightbulbs and add the needed **using** lines.

// GET: Movies/CreateFrom

/\*

\* The user has searched for a movie title.

\* The search results page ("Search.cshtml") has an Add button

\* clicking the add button will take the user to this endpoint

\* which will display the attributes and

\* let the user save the movie to the project database.

\* The save button on the CreateFrom page will send Movie data

\* to the Create method above.

\* Just like any Movie created from the Create link

\*/

public IActionResult CreateFrom([Bind("id,image,title,description")] Movie movie)

{

movie.resultType = "From IMDB";

return View(movie);

}

// POST: Movies/Search/{title}

/\*

\* The user has requested a search of IMDB

\* the application will make an AJAX call to IMDB and

\* return the results (10 movies).

\* text returned is just text.

\* Use JsonSerializer to Deserialize the text to a POCO.

\* The MovieHeader POCO has a few uninteresting attributes

\* plus an array of Movie objects in the results attribute.

\* header.results is the array of Movies

\* The array is passed to the Search page

\* which will list the 10 movies including the movie poster

\*/

[HttpPost]

public IActionResult Search(string title)

{

String baseURL = "https://imdb-api.com/en/API/SearchMovie/k\_XXXXXXX/";

string movieResults = MovieAPI.GetJsonText(baseURL + title);

// Put a break point here and look at the movieResults

// it is a long string of text that looks like JSON

// Deserializer will recognize the text as a MovieHeader object

// WOW! this line of code does very cool stuff!!!

MovieHeader header = JsonSerializer.Deserialize<MovieHeader>(movieResults);

return View(“Index”, header.results);

}

# Update the database

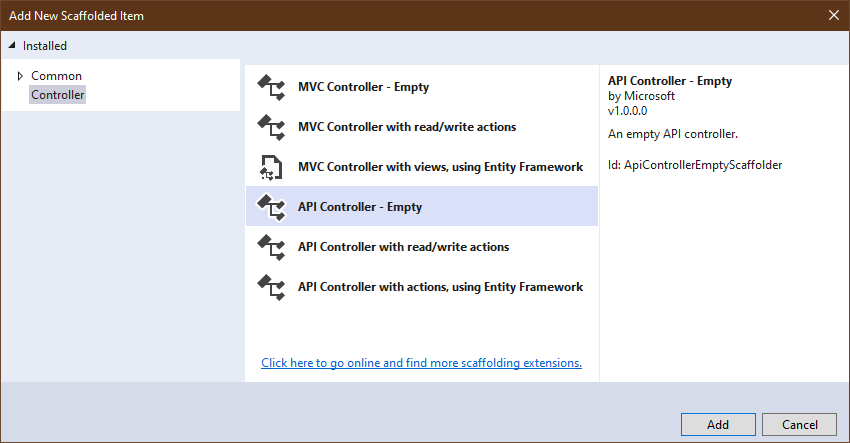
Migrate our new entities to the database.

add-migration -Context MovieAPIContext movies

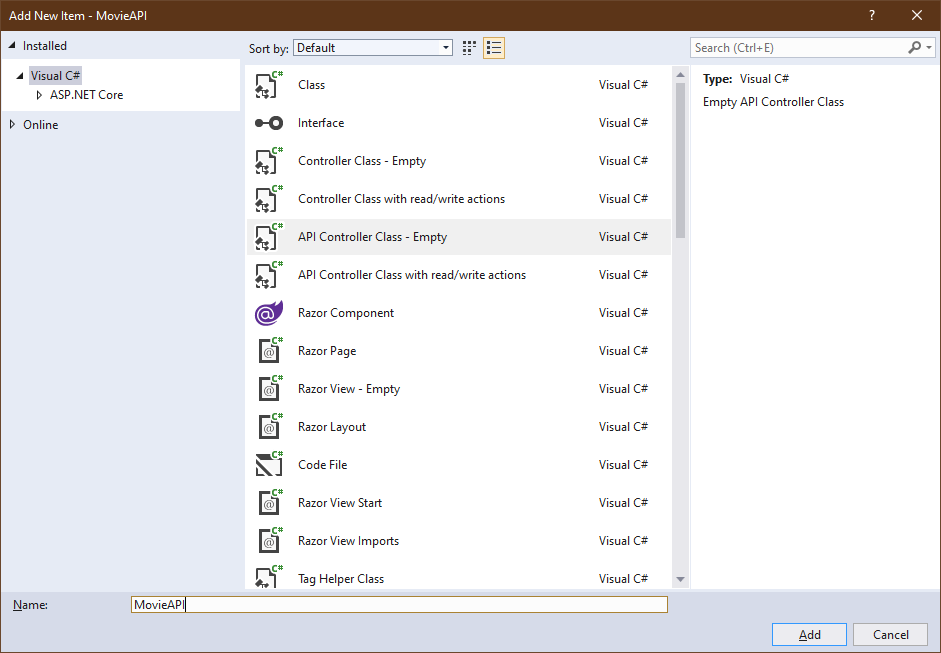
update-database -Context MovieAPIContext

# Adding the Movie API

We need to create an API Controller – Empty



Name the Controller MovieAPI. Make sure that API Controller Class – Empty is selected/



Paste this code into the empty curly brackets for the class. Add your own key value into the highlighted text.

// We just have a single API endpoint here.

// provide a movie title and a list of JSON movies is return

// GET: api/MovieApi/movie/inception

[HttpGet("movie/{title}")]

public ActionResult<Movie[]> GetMovie(String title)

{

String baseURL = "https://imdb-api.com/en/API/SearchMovie/k\_XXXXXXXX/";

// we are using the static MoviesController method GetJsonText to

// hit the IMDB movie API to get our list of movies (DRY)

string movieResults = GetJsonText(baseURL + title);

MovieHeader header = JsonSerializer.Deserialize<MovieHeader>(movieResults);

return header.results;

}

// Returns JSON string

/\*

\* This little piece of software magic will make an AJAX call

\* to the API passed in

\* The request completes and the GetResponseStream method

\* will provide the code with the text from the API

\*/

public static string GetJsonText(string url)

{

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);

try

{

WebResponse response = request.GetResponse();

using (Stream responseStream = response.GetResponseStream())

{

StreamReader reader = new StreamReader(responseStream, System.Text.Encoding.UTF8);

return reader.ReadToEnd();

}

}

catch (WebException ex)

{

// catch if the API request fails

// Log it if you like with the description of the error

// then throw the error (which will kill our application

WebResponse errorResponse = ex.Response;

using (Stream responseStream = errorResponse.GetResponseStream())

{

StreamReader reader = new StreamReader(responseStream, System.Text.Encoding.GetEncoding("utf-8"));

String errorText = reader.ReadToEnd();

// log errorText if you want to

}

throw;

}

}

# Add Search to Movie listing page

Open the Index.cshtml page under Views and under Movies.

Paste this code below the <p> </p> tag block that has the Create New link

<form asp-action="Search">

<div class="form-group">

<label for="title" class="control-label">Title</label>

<input name="title" class="form-control" />

</div>

<div class="form-group">

<input type="submit" value="Search" class="btn btn-primary" />

</div>

</form>

There is a field we get from IMDB resultType. It is not interesting, so remove it from the HTML. It is the first column for the thead and tbody sections of the table. Remove the <th> and <td> sections from <thead> and <tbody> respectively for the resultType attribute

<th> -- down in the tbody this is a <td> tag

@Html.DisplayNameFor(model => model.resultType)

</th>

Now we want to change the image. We have a link to the movie poster. We do not want to show text it will be nicer to show the image itself. Replace this line in Index.cshtml

@Html.DisplayFor(modelItem => item.image)

With this HTML code

<a href="http://www.imdb.com/title/@item.id">

<img src="@item.image" width="75px" />

</a>

This will show the image and let us click on it to take us to the IMDB page for that movie

While we are in the mode of changing HTML files lets add a link to the nav bar or our Movie pages. The file we want to change is in Views -> Shared->\_Layout.cshtml. Find the <li> tags around lines 22-27. Add this block of HTML below the second closing </li> tag

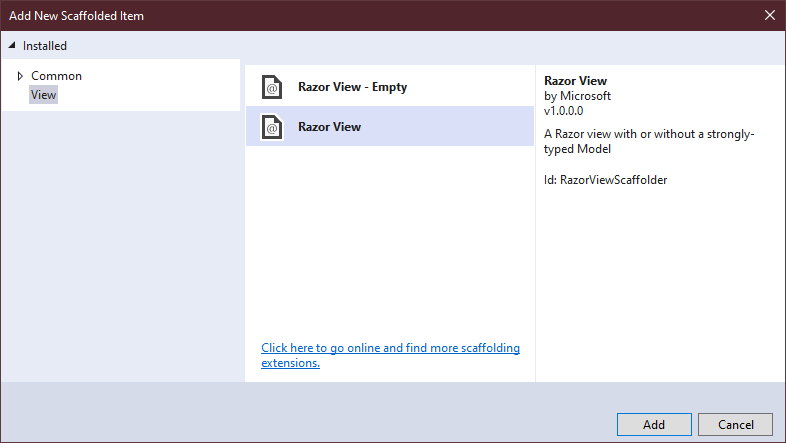
<li class="nav-item">

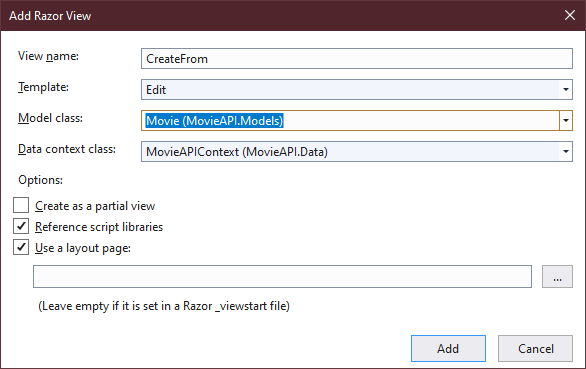
<a class="nav-link text-dark" asp-area="" asp-controller="Movies" asp-action="index">Movies</a>

</li>

# HTML Page to save IMDB Movie to our database

We need one HTML file to complete our application. When we complete our Search we have the option of saving an IMDB movie to our database. We have an Edit page that is good for updating a Movie record IF you have a Movie ID, which we won’t have until we save the movie. We have a Create page that is great for creating a movie from nothing, but we have data to use from IMDB. So we need to create a new page where we are Creating a Movie From existing data. So mostly it looks like an Edit page. Right click on Views->Movies and select Add->View. We will Create a Razor View from a Model class.





Voila we have a page that will be populated with data from IMDB.

* Remove the div for the resultType attribute
* Change the form action to **Create** (not CreateForm).
* Change the H1 tag to something nicer like **Create a Movie From IMDB Data**

# Changes to our web pages

When the search results are displayed, we are going to reuse the Index page, since the purpose of the index page is to display an array Movies it works for our result set from IMDB and our database request.

We will make a small change to the Index page for the database records we want to be able to Display | Edit | Delete them. We need those links to be available. When we do an IMDB search, we might want to **add** a movie to our database so in that situation we will want just an Add link.

See the three links at the bottom of Index.cshtml? We want to replace them with this

@if (ViewBag.AddOnly)

{

<a asp-action="CreateFrom" asp-route-id="@item.id" asp-route-image="@item.image" asp-route-description="@item.description" asp-route-title="@item.title">Add</a>

}

else

{

<a asp-action="Edit" asp-route-id="@item.id">Edit | </a>

<a asp-action="Details" asp-route-id="@item.id">Details | </a>

<a asp-action="Delete" asp-route-id="@item.id">Delete</a>

}

# Show different links depending on the source of the data

Finally, we need to put two lines of code into our MoviesController file. Find the Index method towards the top and insert

ViewBag.AddOnly = false;

Before the return

And down at the very bottom in the Search method we need to add

ViewBag.AddOnly = true;

Just before the return

At last, we have it all working

Run the application click on the Movies link in the navigation bar. From there you can search for movies on IMDB. Do a search. Click Add to see the details of the movie. In the Create from IMDB Data page click the save button which will redirect you to the Index page and show you that your movie is now in the database. The final page of this document is a list of APIs you can use to retrieve all of the data we have set up in this application.

Add these endpoints to the end of your URL it will look something like this (your port # will be different:

<https://localhost:44356/>

**HomeController (MVC)**

/ home page

/Home/Privacy GET Privacy notice

**MoviesController (MVC)**

/Movies GET list all movies

/Movies/Edit/tt0034583 GET edit movie

/Movies/Details/tt0038461 GET show movie details

**MovieAPI (API)**

/api/MovieAPI/movie/{title} GET list movies with matching title from IMDB

**Forecast (API)**

/api/forecast GET randomly generated 5 day forecast

**TodoItems (API)** (Postman will be needed for anything besides GET)

/api/TodoItems GET List all to dos

/api/TodoItems/{id} GET READ one to do

/api/TodoItems/{id} DEL DELETE to do

/api/TodoItems/{id} PUT UPDATE a to do

PUT in Postman. Below is the body  
 { "id":2, "name":"order out to chapps", "isComplete":true }

/api/TodoItems POST CREATE a to do item

POST in Postman. Below is the body

{ "name":"make dinner",  "isComplete":true }