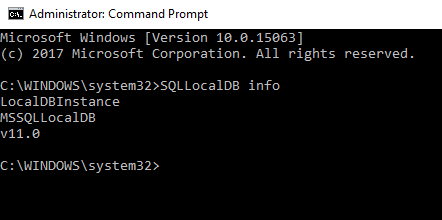
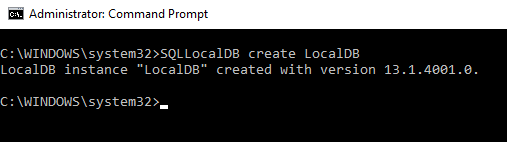
**DOJO – ASP.NET & REACT.JS**

1. **Create a local database**

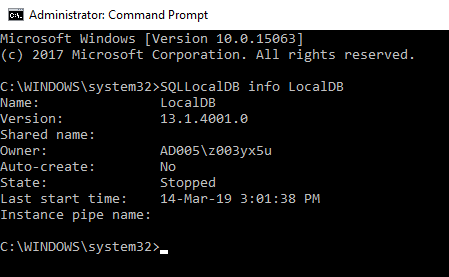
By typing the **SqlLocalDb info** command and pressing the Enter key from the keyboard in the Command Prompt window, the following instances may be found:



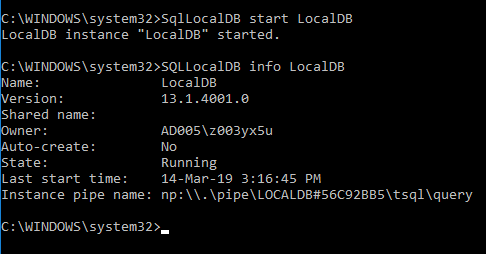
Let’s create a new local db instance. In the Command Prompt window, type the following command: **SqlLocalDB create YourInstanceName** and press the Enter key. The message will appear which indicates that the LocalDB instance is created:



Now, when the **SqlLocalDB info YourInstanceName**  command is executed in the **Auto-create** section, **No** will be shown:



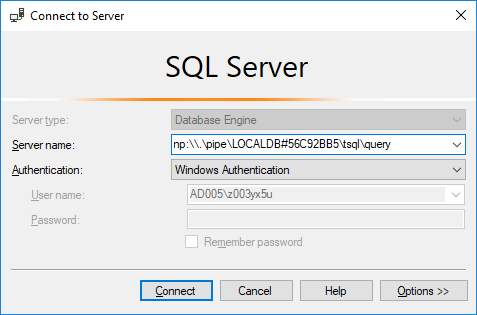
**State** shows a current state of a chosen LocalDB instance, if it is**Running**or it**Stopped.** To run, for example, the Test LocalDB instance, in the Command Prompt window, the following command should be typed: **SqlLocalDB start YourInstanceName** **.**The result after starting the Test LocalDB instance is:



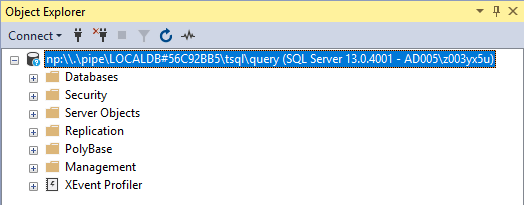
Now, the **State** is changed to “**Running**” and the **Instance pipe name** section has a value. This value (string) is used as the connection string to the Test LocalDB instance from another application.

Now, when an instance is created and started, it is not possible to do other things like creating databases and running queries since SqlLocalDb does not provide an interface to the engine by itself as it does not provide a means to interact with databases. Still, there are several other options to connect to and interact with SqlLocalDb instances.

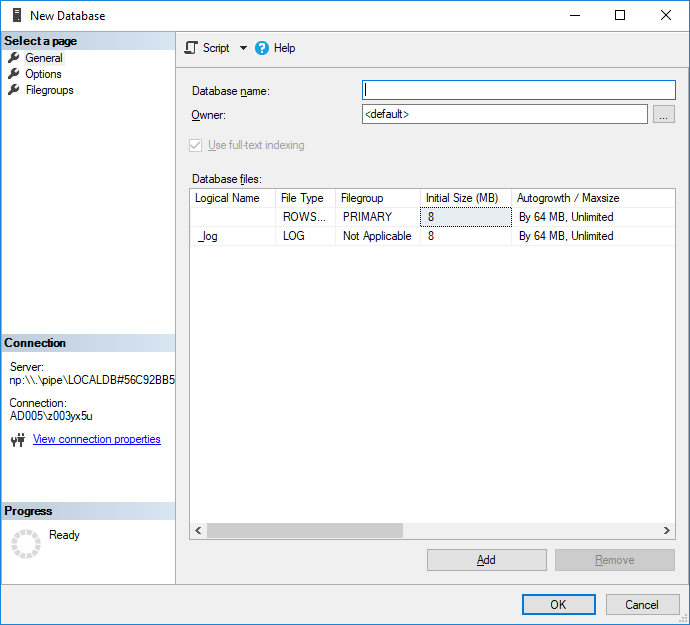
For example, if the string from the **Instance pipe name** is pasted in the Server name field under the Connect to Server window of SQL Server Management Studio:



And the **Connect** button is pressed, the connection to the Test LocalDB instance will be established:



To create new database click on Databases in Object Explorer and then New Database. In the New Database dialog enter your database name and click OK.



1. **Creating database tables**

**#TODO 1**

**Create database tables for given entities and populate them with some data.**

We will be using two tables to store our data.

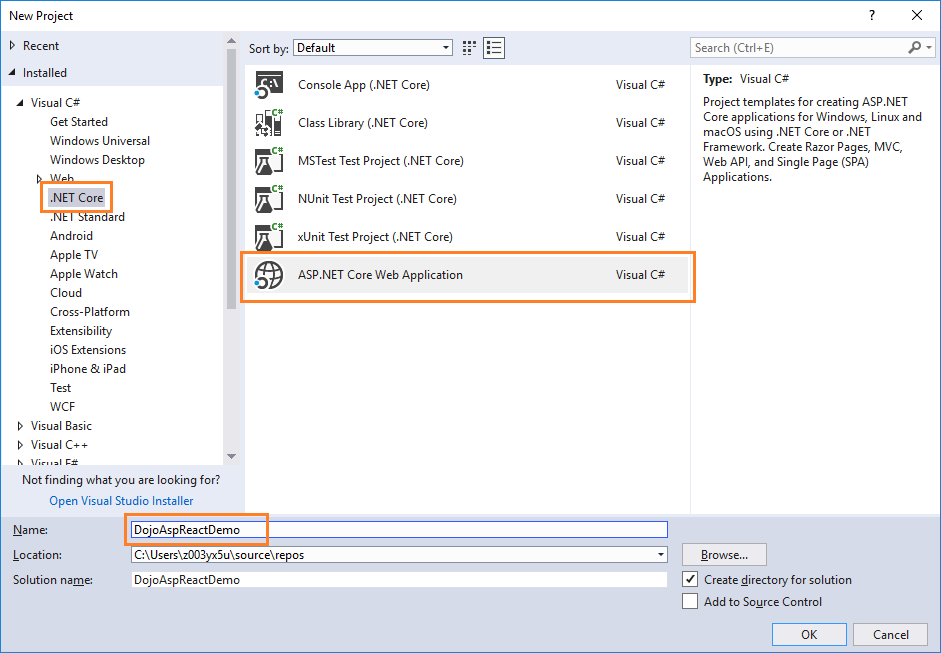
*Patient:*Used to store the details of patients. It contains fields such as Id, Name, Surname, City, Department, and Gender.

*City:*This contains the list of cities and is used to populate the *City* field of the tblPatient table. It contains two fields, Id and CityName.

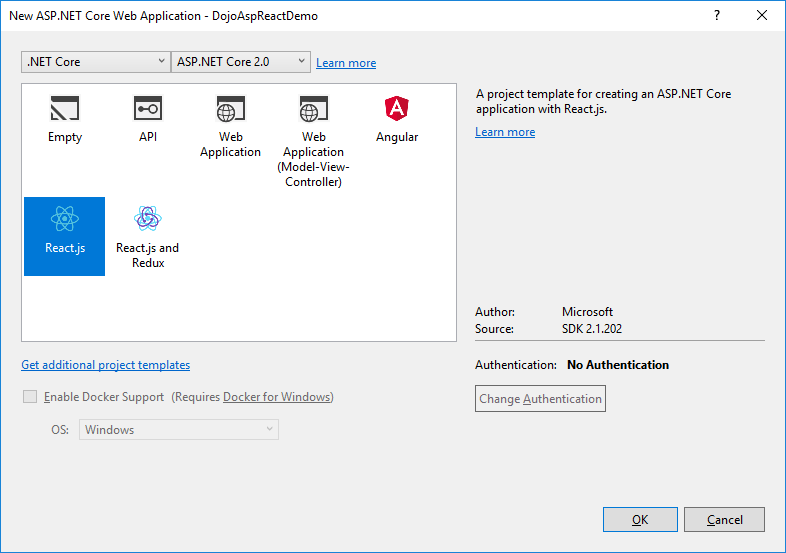
1. **Create an ASP.NET Core Web Application**

Open Visual Studio and select File >> New >> Project. After selecting the project, a "New Project" dialog will open. Select .NET Core inside the Visual C# menu from the left panel.

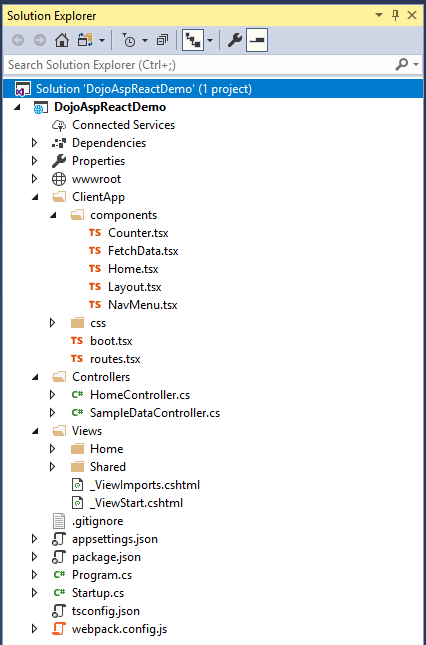
Then, select "ASP.NET Core Web Application" from the available project types. Set the name of the project and press OK.



After clicking on OK, a new dialog will open asking you to select the project template. You can observe two drop-down menus at the top left of the template window. Select ".NET Core" and "ASP.NET Core 2.0" from these dropdowns. Then, select the "React.js" template and press OK.



Now, our project will be created. You can observe the folder structure in Solution Explorer as shown in the below image.



Here, we have our *Controllers* and *Views* folders. We won't be touching the *Views* folders for this tutorial since we will be using React.js to handle the UI. The *Controllers* folders will contain our Web API controller. The point of interest for us is the ClientApp folder where the client side of our application resides. Inside the *ClientApp/components* folder, we already have a few components created which are provided by default with the React.js template in VS 2017. These components will not affect our application, but for the sake of this tutorial, we will delete the *fetchdata.tsx* and *counter.tsx*files from *ClientApp/app/components.*

1. **Adding the model to the Application**

We are using the Entity Framework Core database first approach to create our models. Navigate to Tools >> NuGet Package Manager >> Package Manager Console.

We have to install the package for the database provider that we are targeting, which is SQL Server in this case. Hence, run the following command:

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Since we are using Entity Framework Tools to create a model from the existing database, we will install the tools package as well. Hence, run the following command:

Install-Package Microsoft.EntityFrameworkCore.Tools

After you have installed both the packages, we will scaffold our model from the database tables using the following command:

Scaffold-DbContext "Your connection string here" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Patient, City

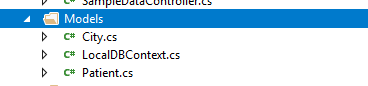
How to build your connection string: <https://www.connectionstrings.com/sql-server/>

F.E.:

Scaffold-DbContext "

Server=MD1SGM6C\LOCALDB#56C92BB5;Database=LocalDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Patient, City

After this command is executed successfully you can observe a Models folder has been created and it contains three class files the Context class*, City.cs,* and *Patient.cs*. Hence, we have successfully created our Models using the EF core database first approach.



**#TODO 2**

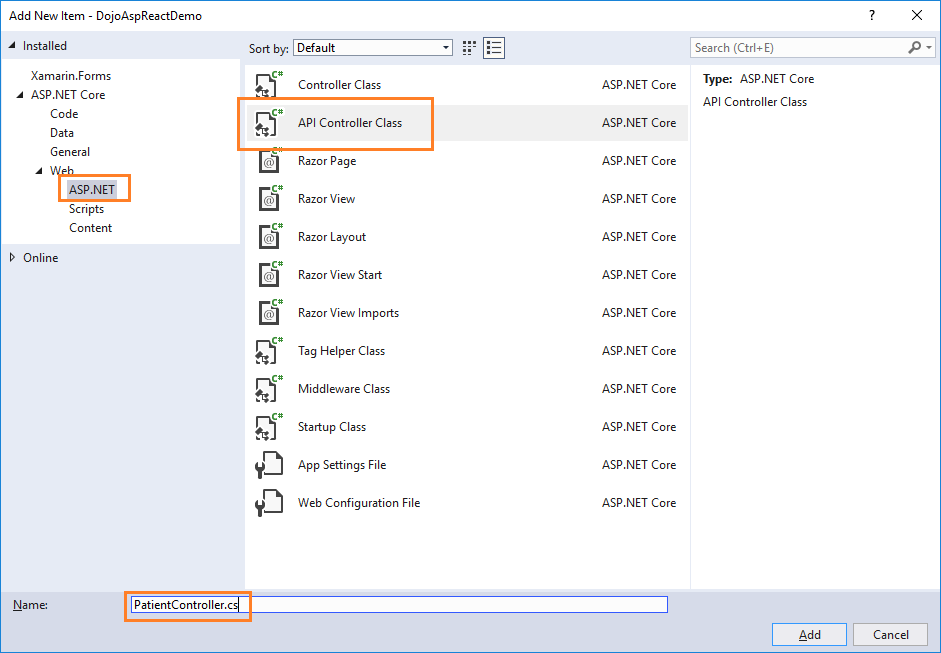
Now we will create one more class file to handle database related operations.

Right click on Models folder and select Add >> Class. Name your class PatientRepository.cs and click Add button. Open PatientRepository.cs and add methods for getting all patients from database, add patient to database, update patient, delete patient and get all cities from database.

1. **Adding the WEB API Controller to the Application**

Right click on Controllers folder and select Add >> New Item.

An "Add New Item" dialog box will open. Select ASP.NET from the left panel, then select "API Controller Class" from the templates panel and set the name as PatientController.cs. Press OK.



This will create our Web API *PatientController*class. We will put all our business logic in this controller. We will call the methods of *PatientRepository* to fetch data and pass on the data to the front-end.

**#TODO 3**

Open the PatientController.cs and put the following code to it.

PatientRepository patientRepository = new PatientRepository();

        [HttpGet]

        [Route("api/Patient/Index")]

        public IEnumerable<Patient> Index()

        {

            return patientRepository. //**TODO**

        }

        [HttpPost]

        [Route("api/Patient/Create")]

        public int Create(Patient patient)

        {

            return patientRepository. //**TODO**

        }

        [HttpGet]

        [Route("api/Patient/Details/{id}")]

        public Patient Details(int id)

        {

            return patientRepository. //**TODO**

        }

        [HttpPut]

        [Route("api/Patient/Edit")]

        public int Edit(Patient patient)

        {

            return patientRepository. //**TODO**

        }

        [HttpDelete]

        [Route("api/Patient/Delete/{id}")]

        public int Delete(int id)

        {

            return patientRepository.DeletePatient(id);

        }

        [HttpGet]

        [Route("api/Patient/GetCityList")]

        public IEnumerable<City> Details()

        {

            return patientRepository. //**TODO**

        }

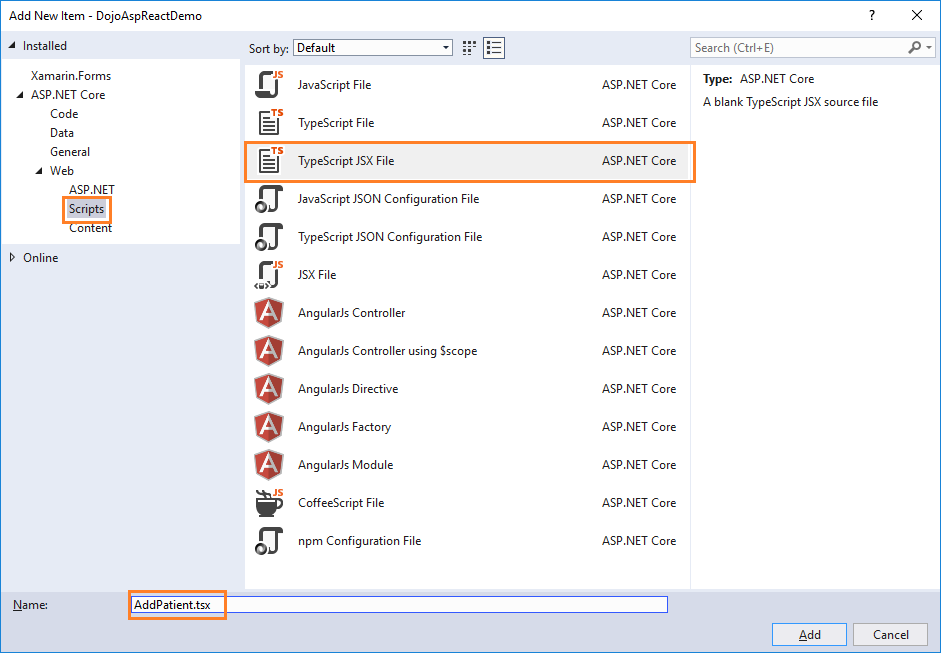
We are done with our backend logic. Therefore, we will now proceed to code our front-end using React.js.

1. **Creating React.js components**

We will be adding two React components to our application:

1. A FetchPatient component - to display all the patient data and delete an existing patient data.
2. An AddPatient component - to add a new patient data and edit an existing patient data.

Right click on *ClientApp/components*folder and select Add >> New Item. An "Add New Item" dialog box will open. Select *Scripts* from the left panel, then select "*TypeScript JSX File*" from the templates panel, and set the name as *AddPatient.tsx*. Press OK. This will add a JSX file inside fo the *components*folder. JSX stand for JavaScript XML. JSX is a preprocessor step that adds XML syntax to JavaScript.



Similarly, add the *FetchPatient.tsx* file to the *ClientApp/components*folder.

1. **FetchPatient Component**

Open the FetchPatient.tsx file and put the following code to it:

import \* as React from 'react';

import { RouteComponentProps } from 'react-router';

import { Link, NavLink } from 'react-router-dom';

interface FetchPatientDataState {

    patList: PatientData[]; // class to hold patient data

    loading: boolean; // indicator if the data is being loaded onto the page

}

const styles = {

    hide: {

        display: 'none',

    }

} as React.CSSProperties;

export class FetchPatients extends React.Component<RouteComponentProps<{}>, FetchPatientDataState> { // component class

    constructor() {

       super();

       this.state = { patList: [], loading: true };

       fetch('api/Patient/Index') // calling WEB API method for fetching all patients

            .then(response => response.json() as Promise<PatientData[]>)

            .then(data => {

                this.setState({ patList: data, loading: false });

            });

        // This binding is necessary to make "this" work in the callback

        this.handleDelete = this.handleDelete.bind(this);

        this.handleEdit = this.handleEdit.bind(this);

    }

    public render() { // render HTML elements into the DOM

        let contents = this.state.loading

            ? <p><em>Loading...</em></p>

            : this.renderPatientTable(this.state.patList);

        return <div>

            <h1>Patient Data</h1>

            <p>This component demonstrates fetching Patient data from the server.</p>

            <p>

                <Link to="/addpatient">Create New</Link>

            </p>

            {contents}

        </div>;

    }

    // Handle Delete request for an Patient

    private handleDelete(id: number) {

        if (!confirm("Do you want to delete patient with Id: " + id))

            return;

        else {

            fetch('api/Patient/Delete/' + id, { // calling WEB API method for fetching all patients

                method: 'delete'

            }).then(data => {

                this.setState(

                    {

                        patList: this.state.patList.filter((rec) => {

                            return (rec.patientId != id);

                        })

                    });

            });

        }

    }

    private handleEdit(id: number) {

        this.props.history.push("/Patient/edit/" + id);

    }

    // Returns the HTML table to the render() method.

    private renderPatientTable(patList: PatientData[]) {

        return <table className='table'>

                   <thead>

                   <tr>

                       <th></th>

                       <th>Name</th>

                       <th>Surname</th>

                       <th>Gender</th>

                       <th>Department</th>

                       <th>City</th>

                   </tr>

                   </thead>

                   <tbody>

                   {patList.map(patient =>

                    <tr key={patient.patientId}>

                        <td></td>

                        <td style={{ display: 'none' }}>{patient.patientId}</td>

                        <td>{patient.name}</td>

                        <td>{patient.surname}</td>

                        <td>{patient.gender}</td>

                        <td>{patient.department}</td>

                        <td>{patient.city}</td>

                        <td>

                            <a className="action" onClick={(id) => this.handleEdit(patient.patientId)}>Edit</a>  |

                            <a className="action" onClick={(id) => this.handleDelete(patient.patientId)}>Delete</a>

                        </td>

                    </tr>

                )}

                   </tbody>

               </table>;

    }

}

export class PatientData {

    patientId: number = 0;

    name: string = "";

    surname: string = "";

    gender: string = "";

    city: string = "";

    department: string = "";

}

Let's understand this code. At the top, we have defined an interface, FetchPatientDataState, which has two properties:

1. patList of type PatientData class to hold the patient data.
2. loading of type boolean to indicate if the data is being loaded onto the page.

After this, we have defined a component class, FetchPatient, which inherits the abstract class React.Component. Inside the constructor of this class, we are calling the base class constructor using Super() and then initializing the fields of the interface to their default values.

We are also calling our web API method usingfetchand setting the patList value and also setting the loading to false. The fetch method is invoked inside the constructor so that patient data will be displayed as the page loads.

At the end of the constructor, we are binding the handleDelete and handleEdit methods. This binding is necessary to make "this" work in the callback.

Then we have our render() method which will render our HTML elements onto the DOM. We will check if the data has finished loading or not and then call the renderPatientTable method which will return an HTML table to display all the patient data on the web page. Every row of the table also has two action methods - Edit and Delete - for editing and deleting the patient records.

Next, we have the handleDelete method which accepts patientID as a parameter. This will prompt the user with a confirmation box and if the user selects 'yes' then it will delete the patient with this patientID.

The handleEdit method will invoke an edit request on the patient record by passing the patient id in the URL parameter and redirects it to the AddPatient component.

At the bottom, we have defined a PatientData class that has the same properties as of our *Patient* model class to hold the patient data.

1. **AddPatient Component**