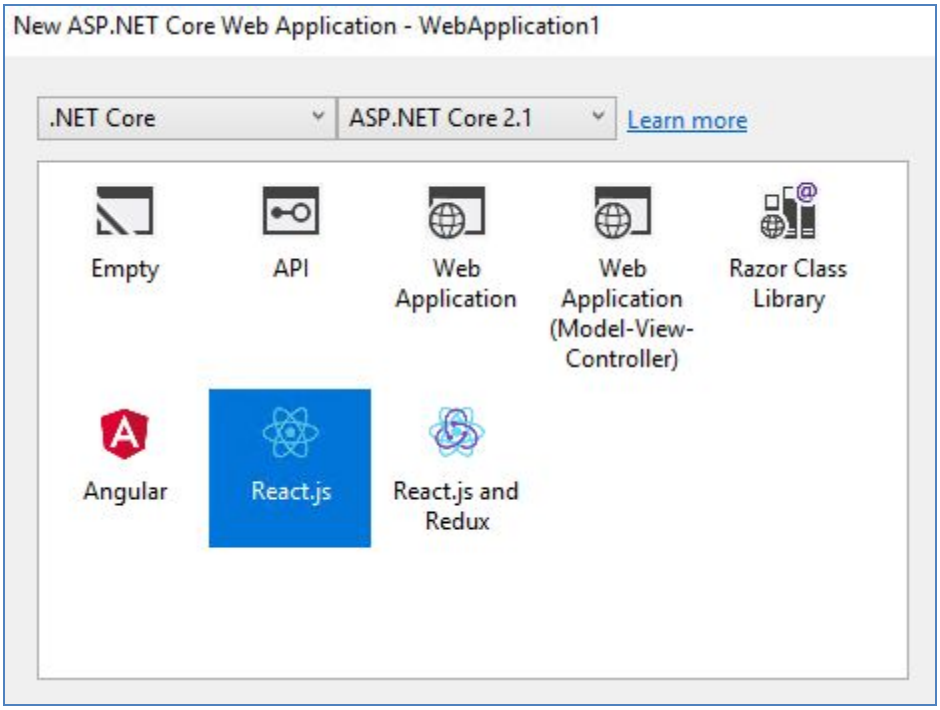
CLASE 7

Laboratorio WebApi con React.js

En este laboratorio veremos cómo crear un webapi .Net Core que sea consumido por un cliente React.js. Tenga en cuenta que dentro de los templates de proyecto de .Net Core, tiene los siguientes:

- Vacio (**Empty**)
- Web Api (**API**)
- Aplicación Web con Razor Pages (**Web Application**)
- Aplicación Web MVC (**Web** Applic**a**tion **Model-View-Controller**)
- Librería de clases Razor (**Razor Class Library**)
- Aplicación Web con Angular (**Angular**)
- Aplicación Web con React.js (**React.js**)
- Aplicación Web con React.js y Redux (**React.js and Redux**)



La aplicación webapi a crear usará **Entity Framework Core** con **Database First**. Crearemos un sistema de gestión de registros de empleados y realizaremos operaciones CRUD en él. Para leer las entradas del usuario, usaremos elementos de formulario HTML con las validaciones de campo requeridas en el lado del cliente. También vamos a vincular una lista desplegable en el formulario HTML a una tabla en la base de datos usando EF Core.

Pasos del ejercicio:

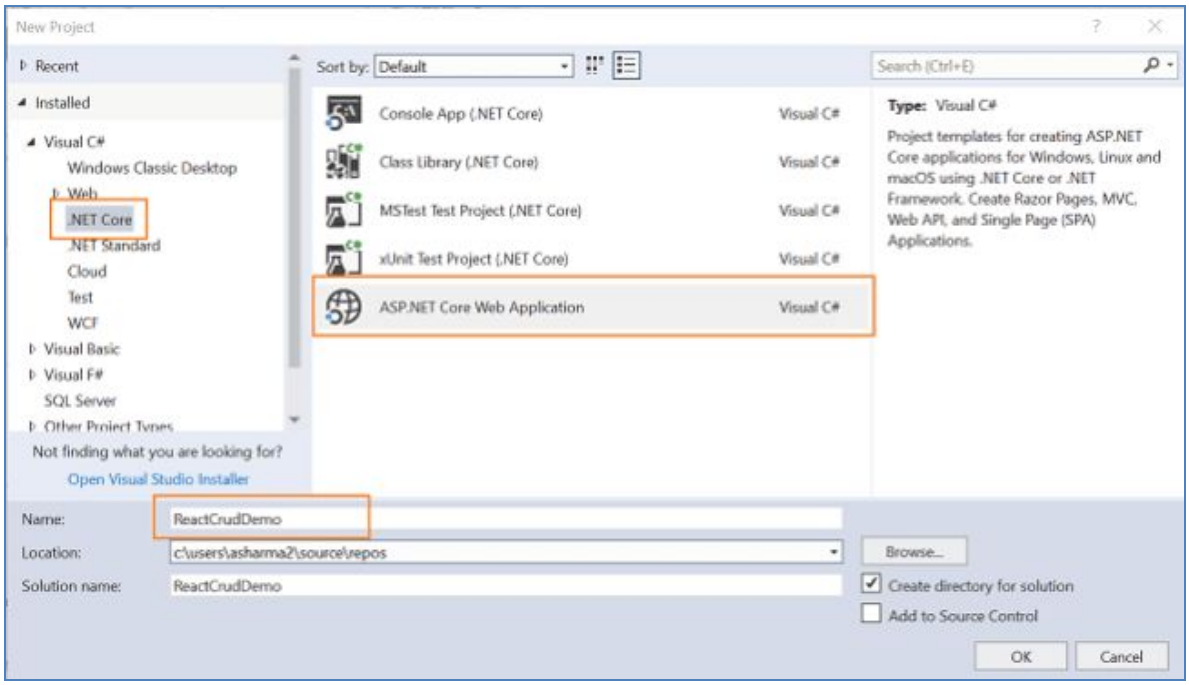
1. Crearemos dos tablas en la base de datos:

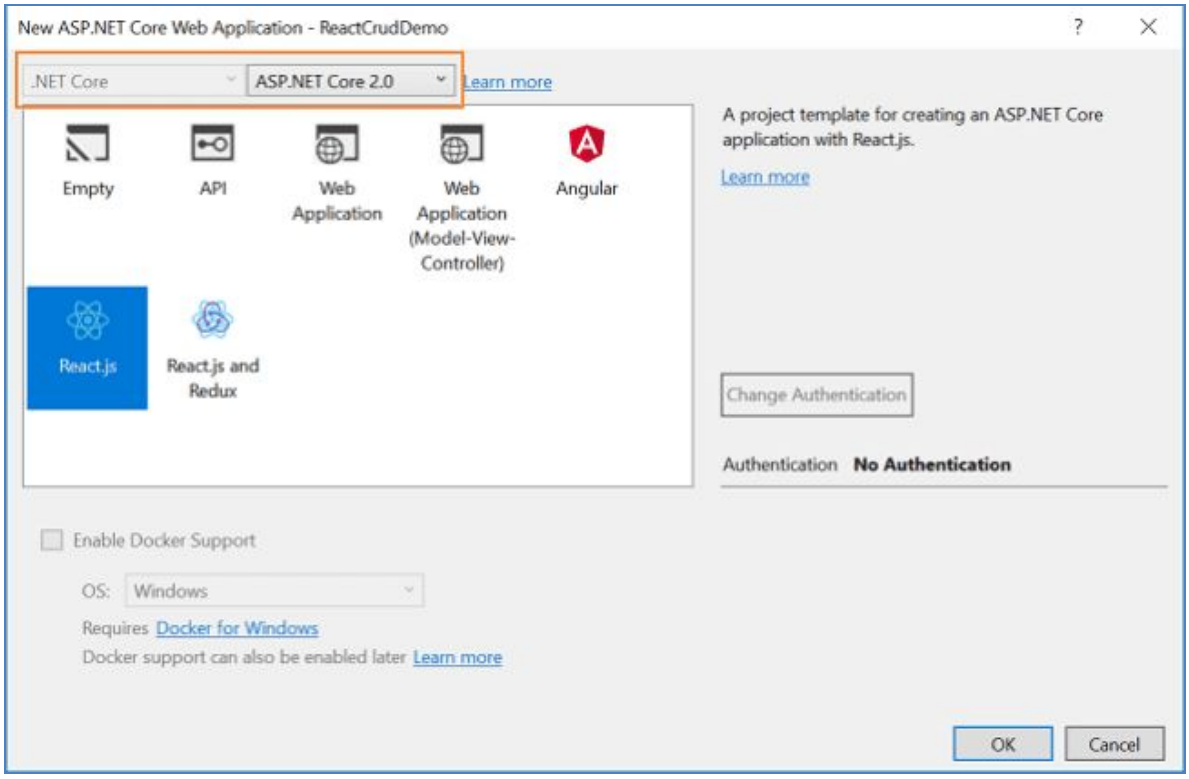
- **tblEmployee**: se utiliza para almacenar los detalles de los empleados. Contiene campos como EmployeeID, Name, City, Department, y Gender.
- **tblCities**: contiene la lista de ciudades y se utiliza para completar el campo Ciudad de la tabla **tblEmployee**. Contiene dos campos, CityID y CityName.

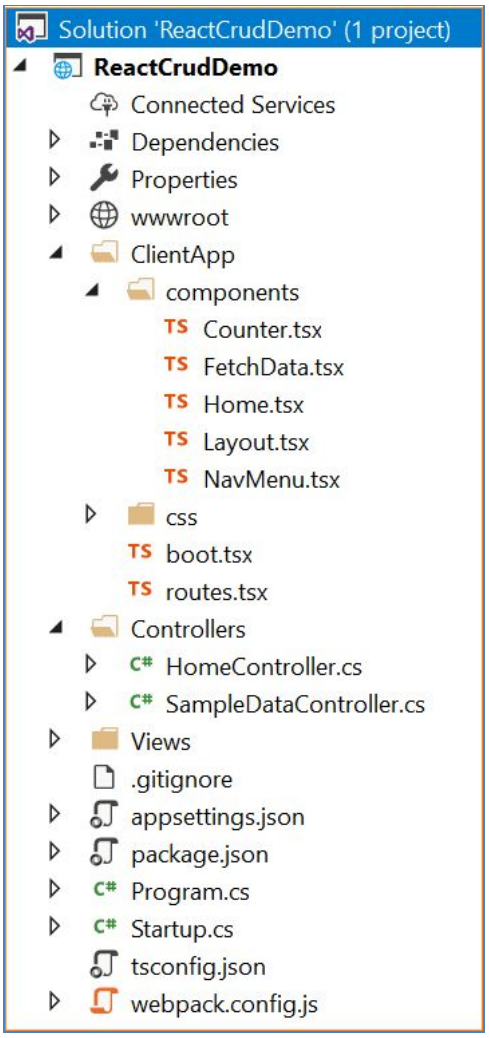
2. Ejecute los siguientes comandos para crear ambas tablas (el archivo de script es **BaseDeDatosCRUDReact.sql** y está en la sección de descargas de la clase)

```
CREATE TABLE tblEmployee (
EmployeeID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
Name varchar(20) NOT NULL ,
City varchar(20) NOT NULL ,
Department varchar(20) NOT NULL ,
Gender varchar(6) NOT NULL
)
GO
CREATE TABLE tblCities (
CityID int IDENTITY(1,1) NOT NULL PRIMARY KEY,
CityName varchar(20) NOT NULL
)
GO

INSERT INTO tblCities VALUES('Rosario');
INSERT INTO tblCities VALUES('Bariloche');
INSERT INTO tblCities VALUES('Nuequén');
INSERT INTO tblCities VALUES('Santa Fé');
INSERT INTO tblCities VALUES('Mar del Plata');
```

3. Creamos un nuevo proyecto WebApi de Asp.Net Core llamado **ReactCrudDemo** con la plantilla de React.js:


New Project
Recent
Installed
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual F#
SQL Server
Other Project Types
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Type: Visual C#
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.
Name: ReactCrudDemo
Location: c:\users\asharma2\source\repos
Solution name: ReactCrudDemo
Browse...
Create directory for solution
Add to Source Control
OK
Cancel


New ASP.NET Core Web Application - ReactCrudDemo
.NET Core
ASP.NET Core 2.0
Learn more
Empty
API
Web Application
Web Application (Model-View-Controller)
Angular
React.js
React.js and Redux
A project template for creating an ASP.NET Core application with React.js.
Learn more
Change Authentication
Authentication No Authentication
Enable Docker Support
OS: Windows
Requires Docker for Windows
Docker support can also be enabled later Learn more
OK
Cancel



El proyecto terminado tendrá la siguiente estructura con los controladores y vistas en sus respectivas carpetas, pero para este ejercicio las vistas (interfaz visual) la manejará React.js.

La carpeta de **Controllers** tendrá los controladores del web api. El punto de interés aquí es la carpeta **ClientApp** donde reside el lado cliente de la aplicación. Dentro de la carpeta **ClientApp/Components**, ya tenemos algunos componentes creados que vienen de forma predeterminada con la plantilla React.js en VS 2017/2019. Estos componentes no afectarán a nuestra aplicación, pero por cuestiones prácticas de este ejercicio eliminaremos los archivos **Fetchdata.tsx** y **Counter.tsx** de **ClientApp/Ccomponents** (o **ClientApp/src/Ccomponents**).

4. Dado que usaremos Entity Framework Core DatabaseFirst, agregaremos el paquete Nuget correspondiente para que se puedan crear las clases del modelo. Seleccione **Tools** - **NuGet Package Manager** - **Package Manager Console**, para instalar el paquete del proveedor de la base de datos, que es SQL Server en este caso. Por lo tanto, ejecute el siguiente comando:

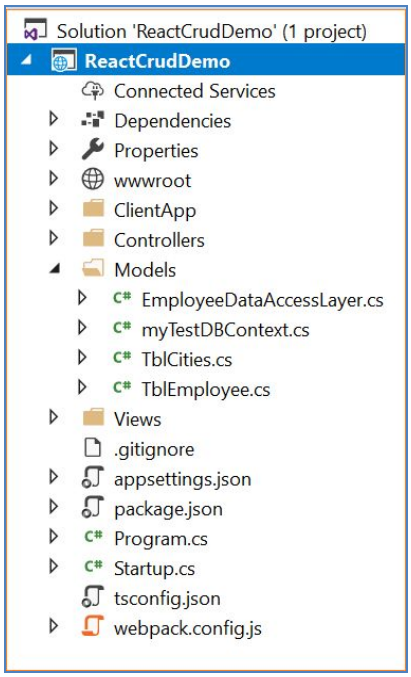
   ```
   Install-Package Microsoft.EntityFrameworkCore.SqlServer
   ```

5. Y también instale el paquete Entity Framework Tools que permitirá crear el modelo desde la base de datos existente:

   ```
   Install-Package Microsoft.EntityFrameworkCore.Tools
   ```

6. Luego de instalar ambos paquetes, deberá ejecutar este comando para hacer un scafolding y generar el modelo:

```
Scaffold-DbContext "Su cadena de conexión va aquí"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables
```



7. No olvide poner tu propia cadena de conexión (dentro de " "). Después de ejecutar este comando correctamente, verá que se ha creado una carpeta **Models** y con tres clases **myTestDBContext.cs**, **TblCities.cs** y **TblEmployee.cs**.

8. Ahora, crearemos una clase más para manejar las operaciones relacionadas con la base de datos.

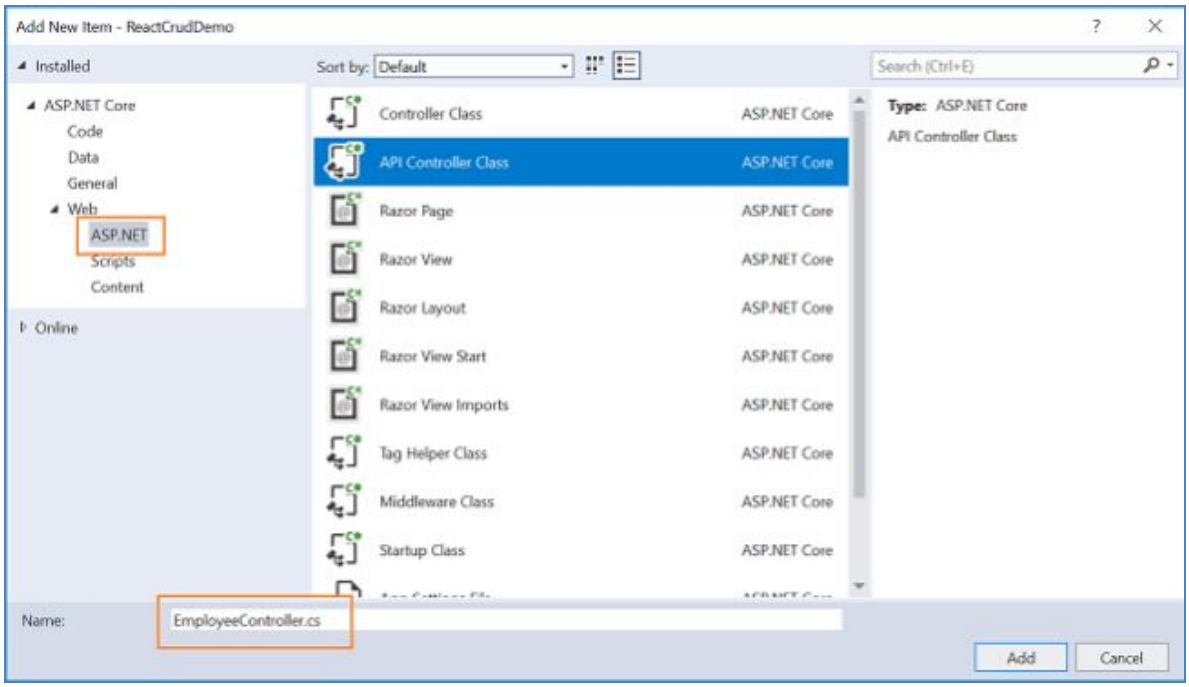
9. Haga clic derecho en la carpeta **Models** y seleccione **Add** - **Class**. Asigne como nombre **EmployeeDataAccessLayer.cs** y haga clic en el botón **Add**. En este momento, la carpeta **Models** tendrá la estructura de la imagen. Aquí irán las operaciones de acceso a la base de datos, es una "capa de acceso a datos" desacoplada del código del controlador.

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ReactCrudDemo.Models
{
    public class EmployeeDataAccessLayer
    {
        myTestDBContext db = new myTestDBContext();

        public IEnumerable<TblEmployee> GetAllEmployees()
        {
            try
            {
                return db.TblEmployee.ToList();
            }
            catch
            {
                throw;
            }
        }

        //To Add new employee record
        public int AddEmployee(TblEmployee employee)
```

```
{
    try
    {
        db.TblEmployee.Add(employee);
        db.SaveChanges();
        return 1;
    }
    catch
    {
        throw;
    }
}

//To Update the records of a particluar employee
public int UpdateEmployee(TblEmployee employee)
{
    try
    {
        db.Entry(employee).State = EntityState.Modified;
        db.SaveChanges();

        return 1;
    }
    catch
    {
        throw;
    }
}

//Get the details of a particular employee
public TblEmployee GetEmployeeData(int id)
{
    try
    {
        TblEmployee employee = db.TblEmployee.Find(id);
        return employee;
    }
    catch
    {
        throw;
    }
}

//To Delete the record of a particular employee
public int DeleteEmployee(int id)
{
    try
    {
```

```
                TblEmployee emp = db.TblEmployee.Find(id);
                db.TblEmployee.Remove(emp);
                db.SaveChanges();
                return 1;
            }
            catch
            {
                throw;
            }
        }

        //To Get the list of Cities
        public List<TblCities> GetCities()
        {
            List<TblCities> lstCity = new List<TblCities>();
            lstCity = (from CityList in db.TblCities select
CityList).ToList();

            return lstCity;
        }

    }
}
```

10. Ahora crearemos el controlador de la api. Para agregar el controlador Haga clic derecho en la carpeta **Controllers** y seleccione **Add** - **New Item**. Se abrirá un cuadro de diálogo "Add New Item ". Seleccione ASP.NET Core en el panel izquierdo, luego seleccione "**Controller class**" en el panel de plantillas e indique como nombre **EmployeeController.cs**. Presiona OK.



11. Esto creará el controlador **EmployeeController**. Pondremos toda la lógica de negocio en este controlador. Llamaremos a los métodos de **EmployeeDataAccessLayer** para obtener datos y pasarlos al front-end. En la clase **EmployeeController.cs** y coloque el siguiente código:

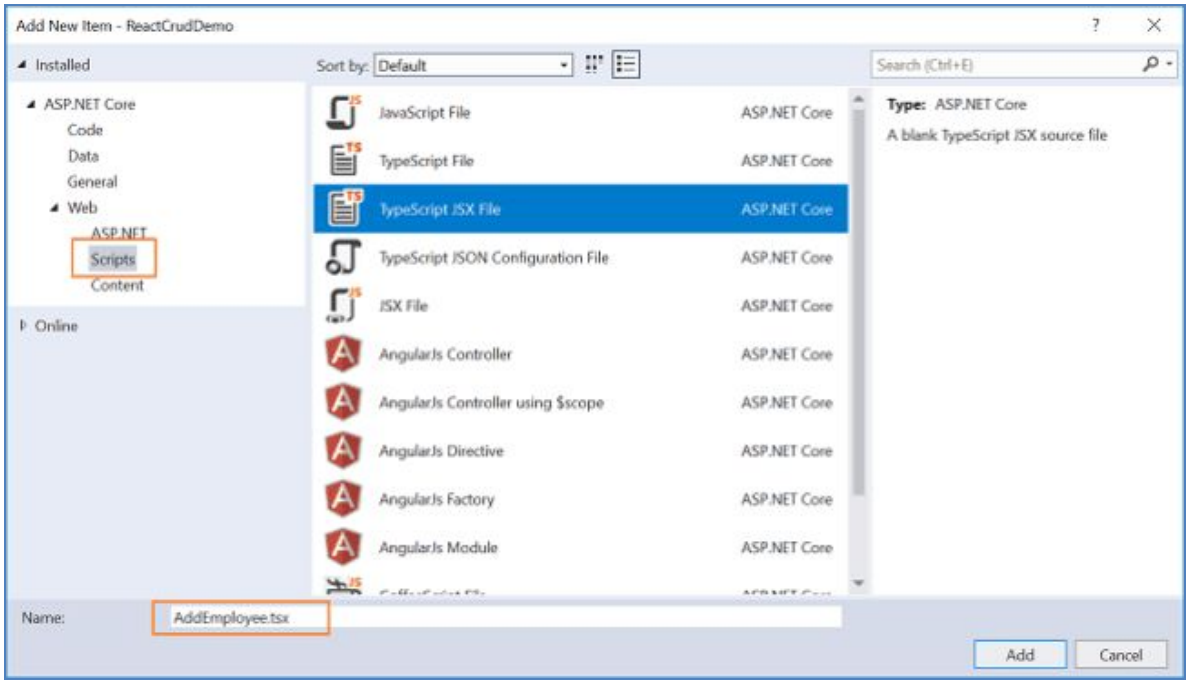
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using ReactCrudDemo.Models;

// For more information on enabling Web API for empty projects, visit
https://go.microsoft.com/fwlink/?LinkID=397860

namespace ReactCrudDemo.Controllers
{
    public class EmployeeController : Controller
    {
        EmployeeDataAccessLayer objemployee = new
EmployeeDataAccessLayer();

        [HttpGet]
        [Route("api/Employee/Index")]
        public IEnumerable<TblEmployee> Index()
```

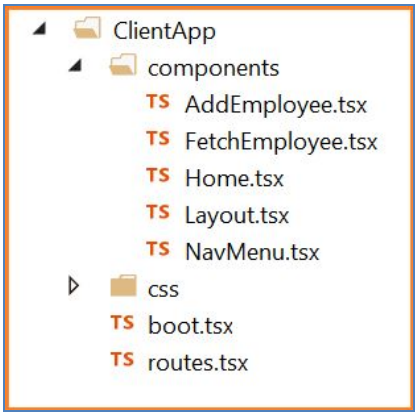
```
        {
            return objemployee.GetAllEmployees();
        }

        [HttpPost]
        [Route("api/Employee/Create")]
        public int Create(TblEmployee employee)
        {
            return objemployee.AddEmployee(employee);
        }

        [HttpGet]
        [Route("api/Employee/Details/{id}")]
        public TblEmployee Details(int id)
        {
            return objemployee.GetEmployeeData(id);
        }

        [HttpPut]
        [Route("api/Employee/Edit")]
        public int Edit(TblEmployee employee)
        {
            return objemployee.UpdateEmployee(employee);
        }

        [HttpDelete]
        [Route("api/Employee/Delete/{id}")]
        public int Delete(int id)
        {
            return objemployee.DeleteEmployee(id);
        }

        [HttpGet]
        [Route("api/Employee/GetCityList")]
        public IEnumerable<TblCities> Details()
        {
            return objemployee.GetCities();
        }
    }
}
```

12. Hemos terminado con nuestra lógica de back-end. Por lo tanto, ahora procederemos a codificar el front-end usando React.js. Para ello agregaremos dos componentes React a la aplicación:

- Un componente **FetchEmployee**: para mostrar todos los datos de los empleados y eliminar los datos de un empleado existente.
- Un componente **AddEmployee**: para agregar los datos de un nuevo empleado y editar los datos de un empleado existente.

13. Haga clic derecho en la carpeta **ClientApp/components** y seleccione **Add – New item**. Se abrirá un cuadro de diálogo "**Add new item**". Seleccione Scripts en el panel izquierdo, luego seleccione " **TypeScript JSX file**" en el panel de plantillas y como nombre escriba **AddEmployee.tsx**. Click en **OK**. Esto agregará un archivo JSX dentro de la carpeta de **components**. JSX significa JavaScript XML. JSX es un paso de preprocesador que agrega sintaxis XML a JavaScript.





14. Del mismo modo, agregue el archivo **FetchEmployee.tsx** a la carpeta **ClientApp/components**. Ahora, la carpeta **ClientApp** tendrá la estructura de la imagen. En **FetchEmployee.tsx** agregue el siguiente código:

```
import * as React from 'react';
import { RouteComponentProps } from 'react-router';
import { Link, NavLink } from 'react-router-dom';

interface FetchEmployeeDataState {
    empList: EmployeeData[];
    loading: boolean;
}

export class FetchEmployee extends
React.Component<RouteComponentProps<{}>, FetchEmployeeDataState> {
    constructor() {
        super();
        this.state = { empList: [], loading: true };

        fetch('api/Employee/Index')
            .then(response => response.json() as Promise<EmployeeData[]>)
            .then(data => {
                this.setState({ empList: 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() {
        let contents = this.state.loading
            ? <p><em>Loading...</em></p>
            : this.renderEmployeeTable(this.state.empList);

        return <div>
            <h1>Employee Data</h1>
            <p>This component demonstrates fetching Employee data from
the server.</p>
            <p>
                <Link to="/addemployee">Create New</Link>
            </p>
            {contents}
        </div>;
    }

    // Handle Delete request for an employee
    private handleDelete(id: number) {
        if (!confirm("Do you want to delete employee with Id: " + id))
            return;
        else {
```

```
            fetch('api/Employee/Delete/' + id, {
                method: 'delete'
            }).then(data => {
                this.setState(
                    {
                        empList: this.state.empList.filter((rec) => {
                            return (rec.employeeId != id);
                        })
                    });
            });
        }
    }

    private handleEdit(id: number) {
        this.props.history.push("/employee/edit/" + id);
    }

    // Returns the HTML table to the render() method.
    private renderEmployeeTable(empList: EmployeeData[]) {
        return <table className='table'>
            <thead>
                <tr>
                    <th></th>
                    <th>EmployeeId</th>
                    <th>Name</th>
                    <th>Gender</th>
                    <th>Department</th>
                    <th>City</th>
                </tr>
            </thead>
            <tbody>
                {empList.map(emp =>
                    <tr key={emp.employeeId}>
                        <td></td>
                        <td>{emp.employeeId}</td>
                        <td>{emp.name}</td>
                        <td>{emp.gender}</td>
                        <td>{emp.department}</td>
                        <td>{emp.city}</td>
                        <td>
                            <a className="action" onClick={(id) =>
this.handleEdit(emp.employeeId)}>Edit</a>  |
                            <a className="action" onClick={(id) =>
this.handleDelete(emp.employeeId)}>Delete</a>
                        </td>
                    </tr>
                )}
            </tbody>
```

```
        </table>;
    }
}

export class EmployeeData {
    employeeId: number = 0;
    name: string = "";
    gender: string = "";
    city: string = "";
    department: string = "";

}
```

15. Explicamos el código anterior. En la parte superior, hemos definido la interfaz **FetchEmployeeDataState**, que tiene dos propiedades:

    - **empList** de tipo clase **EmployeeData** para contener los datos del empleado.
    - **loading** de tipo booleano para indicar si los datos se están cargando en la página.

    Después de esto, se ha definido una clase de componente **FetchEmployee**, que hereda la clase abstracta **React.Component**. Dentro del constructor de esta clase, se llama al constructor de la clase base usando **Super()** y luego inicializando los campos de la interfaz a sus valores predeterminados.

    También estamos llamando al método de web API web **fetch** y configurando el valor **empList** y también configurando la carga en falso. El método **fetch** se invoca dentro del constructor para que los datos de los empleados se muestren a medida que se carga la página.

    Al final del constructor, se vinculan los métodos **handleDelete** y **handleEdit**. Este enlace es necesario para que "esto" funcione en la devolución de llamada.

    Luego tenemos el método **render()** que renderizará los elementos HTML en el DOM. Verificamos si los datos han terminado de cargarse o no y luego llamamos al método **renderEmployeeTable** que devolverá una tabla HTML para mostrar todos los datos de los empleados en la página web. Cada fila de la tabla también tiene dos métodos de acción: Edit y Delete, para editar y eliminar los registros de los empleados.

    A continuación, tenemos el método **handleDelete** que acepta **employeeID** como parámetro. Esto le mostrará al usuario un cuadro de confirmación y si el usuario selecciona 'sí', eliminará al empleado con este Id. El método

handleEdit invocará una solicitud de edición en el registro del empleado pasando la identificación del empleado en el parámetro URL y lo redirigirá al componente **AddEmployee**.

En la parte inferior, hemos definido una clase **EmployeeData** que tiene las mismas propiedades que la clase **TblEmployee** del modelo con los datos del empleado.

16. Abra el archivo **AddEmployee.tsx** y coloque el siguiente código:

```
import * as React from 'react';
import { RouteComponentProps } from 'react-router';
import { Link, NavLink } from 'react-router-dom';
import { EmployeeData } from './FetchEmployee';

interface AddEmployeeDataState {
    title: string;
    loading: boolean;
    cityList: Array<any>;
    empData: EmployeeData;
}

export class AddEmployee extends
React.Component<RouteComponentProps<{}>, AddEmployeeDataState> {
    constructor(props) {
        super(props);

        this.state = { title: "", loading: true, cityList: [],
empData: new EmployeeData };

        fetch('api/Employee/GetCityList')
            .then(response => response.json() as Promise<Array<any>>)
            .then(data => {
                this.setState({ cityList: data });
            });

        var empid = this.props.match.params["empid"];

        // This will set state for Edit employee
        if (empid > 0) {
            fetch('api/Employee/Details/' + empid)
                .then(response => response.json() as
Promise<EmployeeData>)
                .then(data => {
                    this.setState({ title: "Edit", loading: false,
empData: data });
```

```
                });
        }

        // This will set state for Add employee
        else {
            this.state = { title: "Create", loading: false, cityList:
[], empData: new EmployeeData };
        }

        // This binding is necessary to make "this" work in the
callback
        this.handleSave = this.handleSave.bind(this);
        this.handleCancel = this.handleCancel.bind(this);
    }

    public render() {
        let contents = this.state.loading
            ? <p><em>Loading...</em></p>
            : this.renderCreateForm(this.state.cityList);

        return <div>
            <h1>{this.state.title}</h1>
            <h3>Employee</h3>
            <hr />
            {contents}
        </div>;
    }

    // This will handle the submit form event.
    private handleSave(event) {
        event.preventDefault();
        const data = new FormData(event.target);

        // PUT request for Edit employee.
        if (this.state.empData.employeeId) {
            fetch('api/Employee/Edit', {
                method: 'PUT',
                body: data,

            }).then((response) => response.json())
                .then((responseJson) => {
                    this.props.history.push("/fetchemployee");
                })
        }

        // POST request for Add employee.
        else {
            fetch('api/Employee/Create', {
```

```
                method: 'POST',
                body: data,

            }).then((response) => response.json())
                .then((responseJson) => {
                    this.props.history.push("/fetchemployee");
                })
        }
    }

    // This will handle Cancel button click event.
    private handleCancel(e) {
        e.preventDefault();
        this.props.history.push("/fetchemployee");
    }

    // Returns the HTML Form to the render() method.
    private renderCreateForm(cityList: Array<any>) {
        return (
            <form onSubmit={this.handleSave} >
                <div className="form-group row" >
                    <input type="hidden" name="employeeId"
value={this.state.empData.employeeId} />
                </div>
                < div className="form-group row" >
                    <label className=" control-label col-md-12"
htmlFor="Name">Name</label>
                    <div className="col-md-4">
                        <input className="form-control" type="text"
name="name" defaultValue={this.state.empData.name} required />
                    </div>
                </div >
                <div className="form-group row">
                    <label className="control-label col-md-12"
htmlFor="Gender">Gender</label>
                    <div className="col-md-4">
                        <select className="form-control"
data-val="true" name="gender" defaultValue={this.state.empData.gender}
required>
                            <option value="">-- Select Gender
--</option>
                            <option value="Male">Male</option>
                            <option value="Female">Female</option>
                        </select>
                    </div>
                </div >
                <div className="form-group row">
```

```
                    <label className="control-label col-md-12"
htmlFor="Department" >Department</label>
                    <div className="col-md-4">
                        <input className="form-control" type="text"
name="Department" defaultValue={this.state.empData.department}
required />
                    </div>
                </div>
                <div className="form-group row">
                    <label className="control-label col-md-12"
htmlFor="City">City</label>
                    <div className="col-md-4">
                        <select className="form-control"
data-val="true" name="City" defaultValue={this.state.empData.city}
required>
                            <option value="">-- Select City
--</option>
                            {cityList.map(city =>
                                <option key={city.cityId}
value={city.cityName}>{city.cityName}</option>
                            )}
                        </select>
                    </div>
                </div >
                <div className="form-group">
                    <button type="submit" className="btn
btn-default">Save</button>
                    <button className="btn"
onClick={this.handleCancel}>Cancel</button>
                </div >
            </form >
        )
    }

}
```

17. Este componente se usará para agregar y editar los datos de los empleados. Como usaremos la clase **EmployeeData** para almacenar los datos, los hemos importado del componente **FetchEmployee**.

La interfaz **AddEmployeeDataState** tiene cuatro propiedades:

- **title**: para mostrar "Crear" o "Editar" en la parte superior de la página.
- **loading**: para verificar si la página ha terminado de cargar datos.
- **cityList**: para mantener la lista de ciudades de la tabla **tblCities**.

- **empData**: para guardar los datos del empleado para vincularlos al formulario HTML.

Dentro del constructor de la clase de componente **AddEmployee**, se inicializan los campos de la interfaz a su valor predeterminado y luego configurando el valor de la propiedad **CityList** para buscar los datos de la tabla **tblCities**. Usaremos esto para vincular un menú desplegable en el formulario HTML. Como estamos buscando la lista de ciudades dentro del constructor, la lista desplegable se completará a medida que se cargue la página.

Este componente manejará las solicitudes Agregar y Editar. Entonces, ¿cómo va a diferenciar el sistema entre ambas solicitudes? La respuesta es vía ruteo. Necesitamos definir dos parámetros de ruta diferentes, uno para agregar un registro de empleado y otro para editar un registro de empleado. Los definiremos en el archivo **routes.tsx** en breve.

Si se realiza una solicitud de edición, la identificación del empleado se pasará en el parámetro. Dentro del constructor, leeremos el valor del parámetro URL vacío. Si el valor de **empid** es mayor que cero, entonces esta es una solicitud de edición y estableceremos el valor del título en "Editar", llenamos los datos de la propiedad **empData** y configuraremos la carga como falsa.

Si no se establece el valor vacío, entonces es una solicitud de agregado y estableceremos el valor del título en "Crear" y configuraremos la carga en falso.

El método **handleSave** manejará el evento guardar en el formulario. Según si la URL tiene un parámetro vacío o no, enviamos una solicitud PUT o POST y una vez que tenga éxito, redirigimos al usuario al componente **FectchEmployee**.

El método **renderCreateForm** devolverá un formulario HTML que se mostrará en la página web. Hemos establecido el valor predeterminado en todos los campos del formulario. Si se realiza una solicitud Agregar, todos los campos se vaciarán. Si se realiza una solicitud de edición, se completará los datos del empleado correspondiente. Vinculamos el elemento select usando la propiedad **cityList** que hemos poblado en nuestro constructor.

En este momento, es posible que obtenga un error, "*Parameter 'props' implicitly has an 'any' type*"' en el archivo **AddEmployee.tsx**. Si tiene este problema, agregue la siguiente línea dentro del archivo **tsconfig.json:**


```
"noImplicitAny": false
```

18. Para definir la ruta y un menú de navegación de la aplicación, abra el archivo **ClientApp/routes.tsx** y coloque el siguiente código:

```
import * as React from 'react';
import { Route } from 'react-router-dom';
import { Layout } from './components/Layout';
import { Home } from './components/Home';
import { FetchEmployee } from './components/FetchEmployee';
import { AddEmployee } from './components/AddEmployee';

export const routes = <Layout>
    <Route exact path='/' component={Home} />
    <Route path='/fetchemployee' component={FetchEmployee} />
    <Route path='/addemployee' component={AddEmployee} />
    <Route path='/employee/edit/:empid' component={AddEmployee} />

</Layout>;
```

19. En este archivo hemos definido las rutas de la aplicación de la siguiente manera:

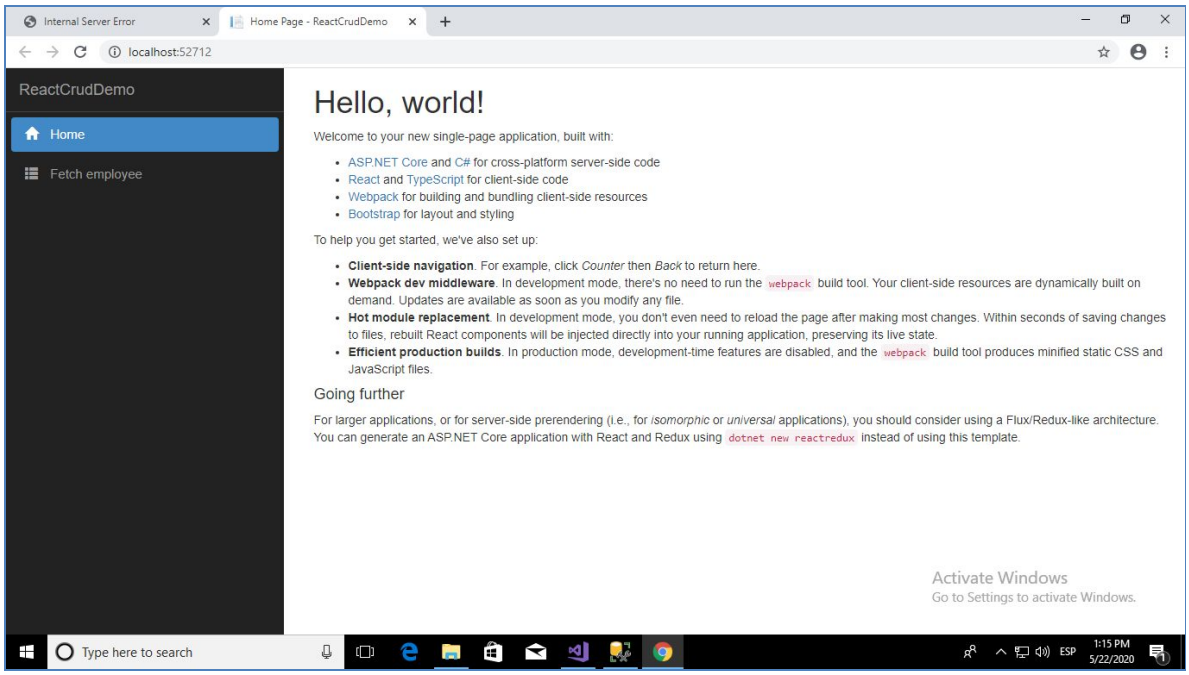
    - **/** si cae en la URL base, se redirigirá al componente **Home**.
    - **/fetchemployee** redirigirá al componente **FetchEmployee**.
    - **addemployee** redirigirá al componente **AddEmployee** para agregar un nuevo registro de empleado.
    - **/employee/edit/:empid** estamos pasando la identificación del empleado en el parámetro. Redirigirá al componente **AddEmployee** para editar los datos de los empleados existentes.

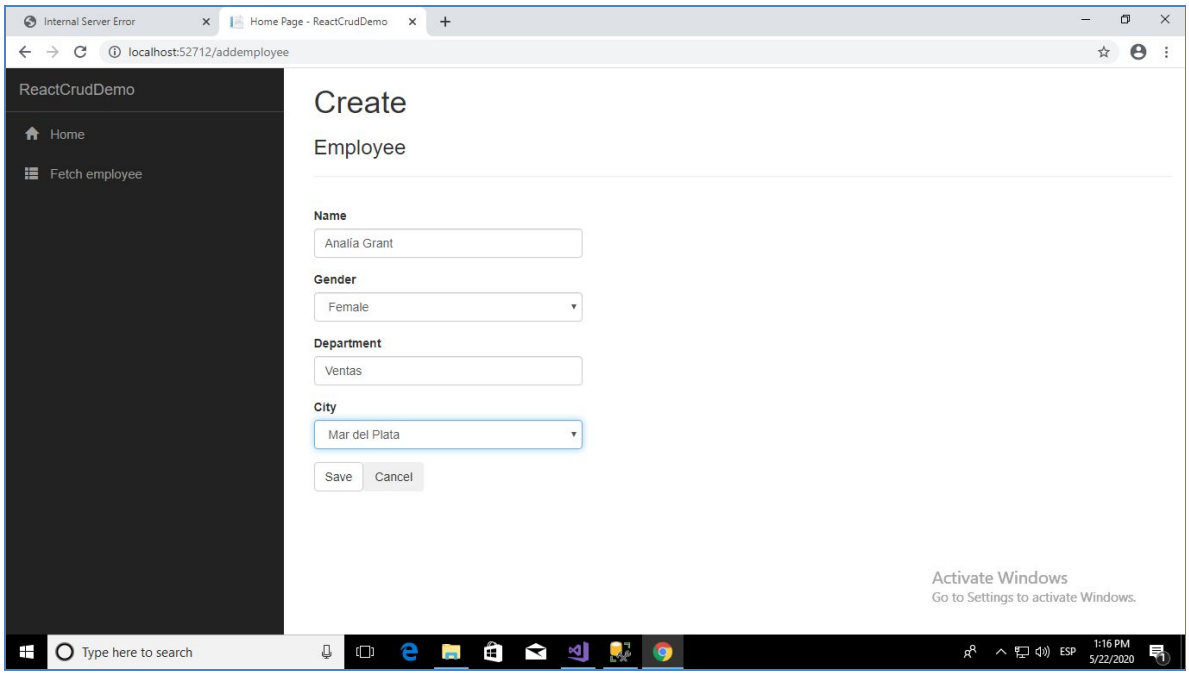
20. Lo último que queda es definir el menú de navegación de la aplicación. Abra el archivo **ClientApp/componentsNavMenu.tsx** y coloque el siguiente código:.

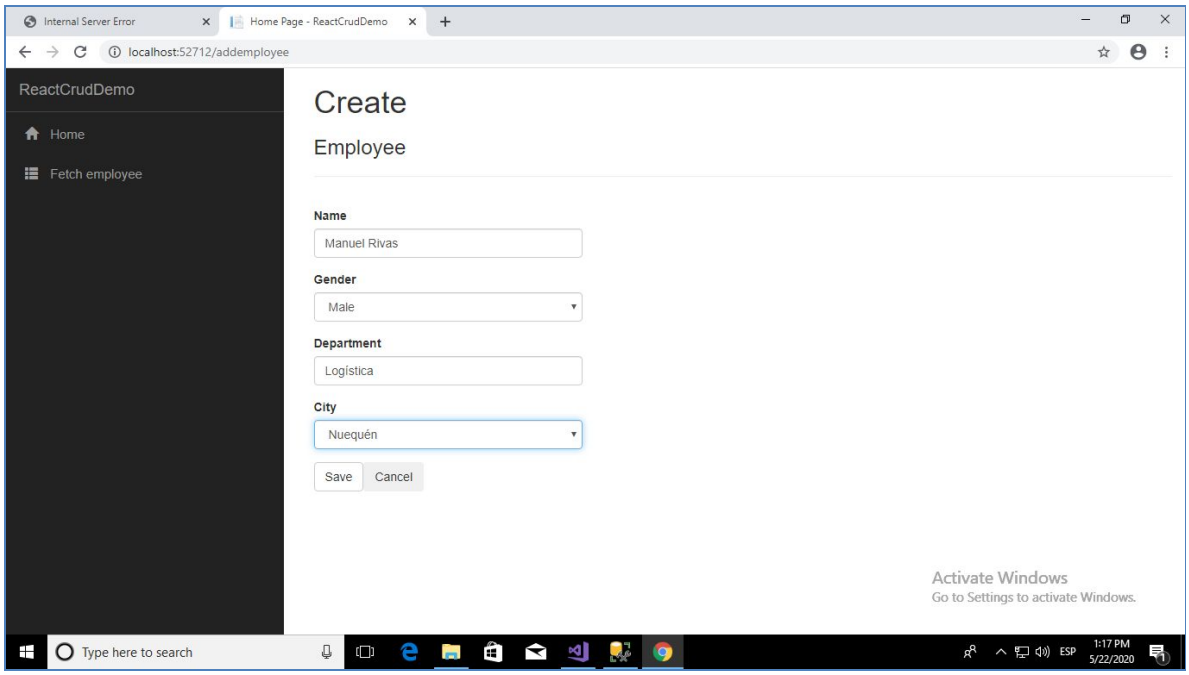
```
import * as React from 'react';
import { Link, NavLink } from 'react-router-dom';

export class NavMenu extends React.Component<{}, {}> {
    public render() {
        return <div className='main-nav'>
            <div className='navbar navbar-inverse'>
                <div className='navbar-header'>
                    <button type='button' className='navbar-toggle'
data-toggle='collapse' data-target='.navbar-collapse'>
                        <span className='sr-only'>Toggle
navigation</span>
                        <span className='icon-bar'></span>
                        <span className='icon-bar'></span>
```

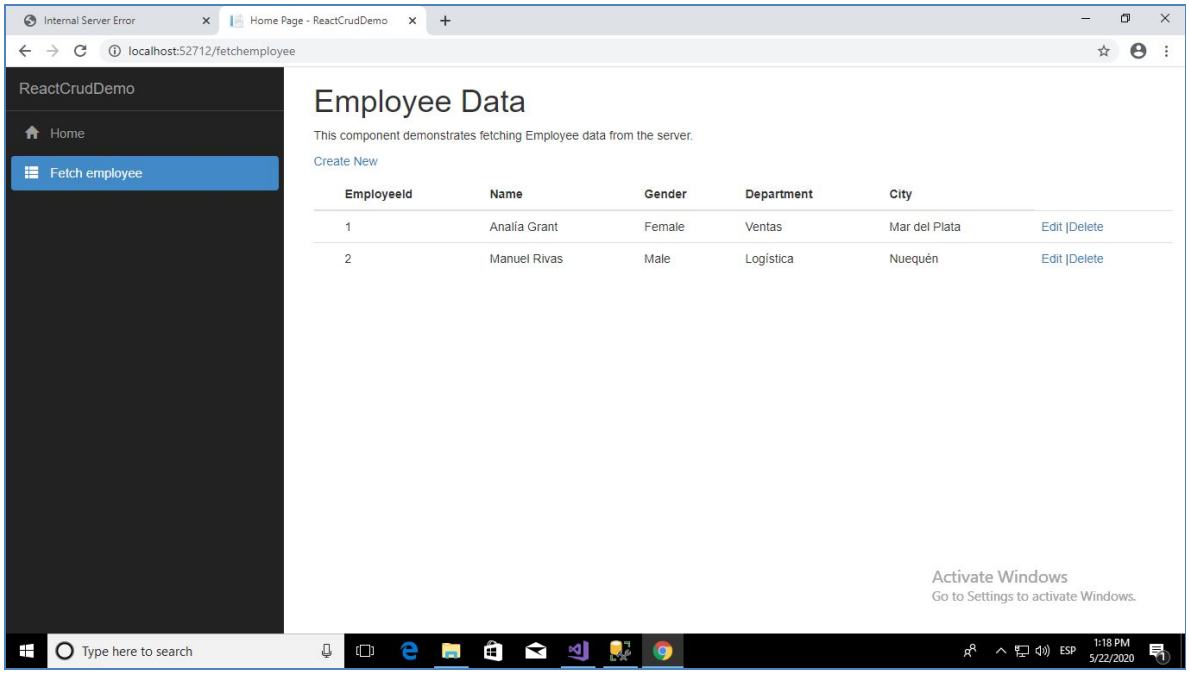
```
                        <span className='icon-bar'></span>
                    </button>
                    <Link className='navbar-brand'
to={'/'}>ReactCrudDemo</Link>
                </div>
                <div className='clearfix'></div>
                <div className='navbar-collapse collapse'>
                    <ul className='nav navbar-nav'>
                        <li>
                            <NavLink to={'/'} exact
activeClassName='active'>
                                <span className='glyphicon
glyphicon-home'></span> Home
                            </NavLink>
                        </li>
                        <li>
                            <NavLink to={'/fetchemployee'}
activeClassName='active'>
                                <span className='glyphicon
glyphicon-th-list'></span> Fetch employee
                            </NavLink>
                        </li>
                    </ul>
                </div>
            </div>
        </div>;
    }
}
```

21. Hemos creado la aplicación ASP.NET Core utilizando React.js y con Entity Framework Core Database First. Presione F5 para iniciar la aplicación. Se abrirá una página web como se muestra en la imagen a continuación. Observe el menú de navegación a la izquierda que muestra el enlace de navegación para las páginas Inicio y Obtener empleados. Pruebe las operaciones del CRUD.


Internal Server Error
Home Page - ReactCrudDemo
localhost:52712
ReactCrudDemo
Home
Fetch employee
Hello, world!
Welcome to your new single-page application, built with:
ASP.NET Core and C# for cross-platform server-side code
React and TypeScript for client-side code
Webpack for building and bundling client-side resources
Bootstrap for layout and styling
To help you get started, we've also set up:
Client-side navigation. For example, click Counter then Back to return here.
Webpack dev middleware. In development mode, there's no need to run the webpack build tool. Your client-side resources are dynamically built on demand. Updates are available as soon as you modify any file.
Hot module replacement. In development mode, you don't even need to reload the page after making most changes. Within seconds of saving changes to files, rebuilt React components will be injected directly into your running application, preserving its live state.
Efficient production builds. In production mode, development-time features are disabled, and the webpack build tool produces minified static CSS and JavaScript files.
Going further
For larger applications, or for server-side prerendering (i.e., for isomorphic or universal applications), you should consider using a Flux/Redux-like architecture. You can generate an ASP.NET Core application with React and Redux using dotnet new reactredux instead of using this template.
Activate Windows
Go to Settings to activate Windows.
Type here to search
ESP
1:15 PM
5/22/2020


Internal Server Error
Home Page - ReactCrudDemo
localhost:52712/addemployee
ReactCrudDemo
Home
Fetch employee
Create
Employee
Name
Analía Grant
Gender
Female
Department
Ventas
City
Mar del Plata
Save
Cancel
Activate Windows
Go to Settings to activate Windows.
Type here to search
ESP
1:16 PM
5/22/2020





El proyecto completo se encuentra en la sección de descarga de la clase.