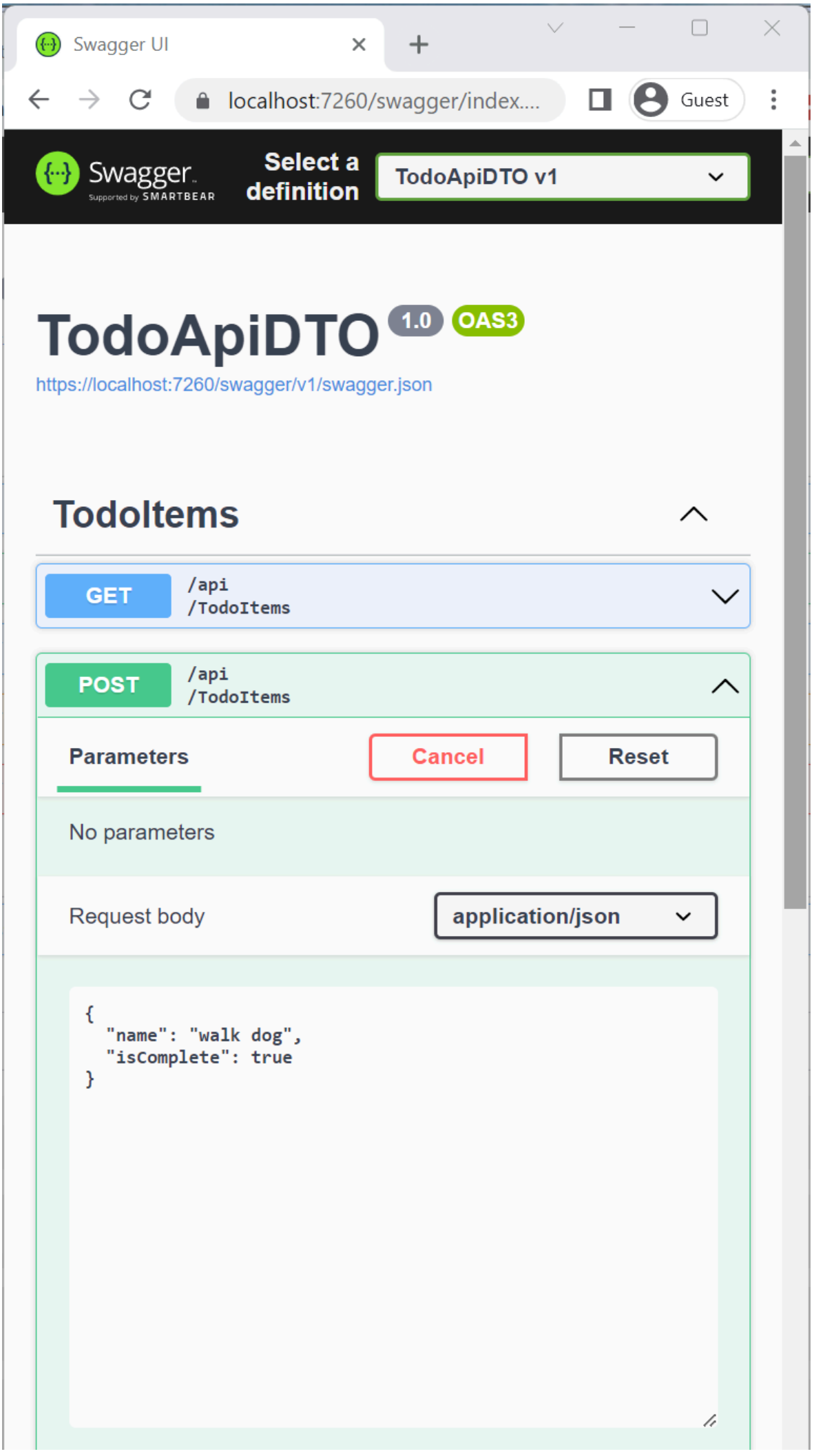

Swagger UI
localhost:7260/swagger/index....
Guest
Swagger
Supported by SMARTBEAR
Select a definition
TodoApiDTO v1
TodoApiDTO
1.0
OAS3
https://localhost:7260/swagger/v1/swagger.json
TodoItems
GET
/api
/TodoItems
POST
/api
/TodoItems
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"name": "walk dog",
"isComplete": true
}

Test the location header URI

In the preceding POST, the Swagger UI shows the location header under **Response headers**. For example, `location: https://localhost:7260/api/TodoItems/1`. The location header shows the URI to the created resource.

To test the location header:

- In the Swagger browser window, select **GET /api/TodoItems/{id}**, and then select **Try it out**.
- Enter `1` in the `id` input box, and then select **Execute**.

GET /api/TodoItems/{id}

Parameters Cancel

Name	Description
id * required integer($int64) (path)	1

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'https://localhost:7260/api/TodoItems/1' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7260/api/TodoItems/1
```

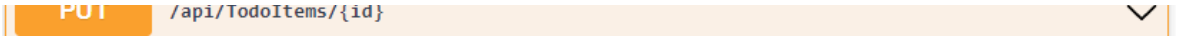
Server response

Code	Details

200	Response body
	`{ "id": 1, "name": "walk dog", "isComplete": true }` Download
	Response headers
	`content-type: application/json; charset=utf-8`
	`date: Fri,21 Oct 2022 00:32:45 GMT`
	`server: Kestrel`

Responses

Code	Description	Links
200	Success Media type: text/plain Controls Accept header. Example Value | Schema `{ "id": 0, "name": "string", "isComplete": true }`	*No links*



Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

The previous section showed an example of the `/api/todoitems/{id}` route.

Follow the POST instructions to add another todo item, and then test the `/api/todoitems` route using Swagger.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request doesn't return any data. If no data is returned, POST data to the app.

Routing and URL paths

The [HttpGet] attribute denotes a method that responds to an `HTTP GET` request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
C#

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.

- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an `HTTP 200` response.

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
```

```
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses `HTTP PUT`. The response is 204 (No Content). According to the HTTP specification, a `PUT` request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Using the Swagger UI, use the PUT button to update the `TodoItem` that has Id = 1 and set its name to `"feed fish"`. Note the response is HTTP 204 No Content.

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
```

```
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use the Swagger UI to delete the `TodoItem` that has Id = 1. Note the response is HTTP 204 No Content.

Test with other tools

There are many other tools that can be used to test web APIs, for example:

- Visual Studio Endpoints Explorer and .http files
- http-repl
- curl. Swagger uses `curl` and shows the `curl` commands it submits.
- Fiddler

For more information, see:

- Minimal API tutorial: test with .http files and Endpoints Explorer
- Install and test APIs with http-repl

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this tutorial.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

C#

```
namespace TodoApi.Models;

public class TodoItemDTO
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers;

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
```

```
    {
        _context = context;
    }

    // GET: api/TodoItems
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    // GET: api/TodoItems/5
    // <snippet_GetByID>
    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }
    // </snippet_GetByID>

    // PUT: api/TodoItems/5
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Update>
    [HttpPut("{id}")]
    public async Task<IActionResult> PutTodoItem(long id, TodoItemDTO
todoDTO)
    {
        if (id != todoDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoDTO.Name;
        todoItem.IsComplete = todoDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
```

```
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }
    // </snippet_Update>

    // POST: api/TodoItems
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Create>
    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> PostTodoItem(TodoItemDTO
todoDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoDTO.IsComplete,
            Name = todoDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
    // </snippet_Create>

    // DELETE: api/TodoItems/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id)
    {
        return _context.TodoItems.Any(e => e.Id == id);
    }

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
```

```
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

Web API video series

See Video: Beginner's Series to: Web APIs.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see **Migrate from ASP.NET Core 5.0 to 6.0**.

For more information, see the Duende Identity Server documentation (Duende Software website).

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Additional resources

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- Tutorial: Create a minimal API with ASP.NET Core
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- Create a web API with ASP.NET Core

Create a web API with ASP.NET Core and MongoDB

Article • 04/25/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that runs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

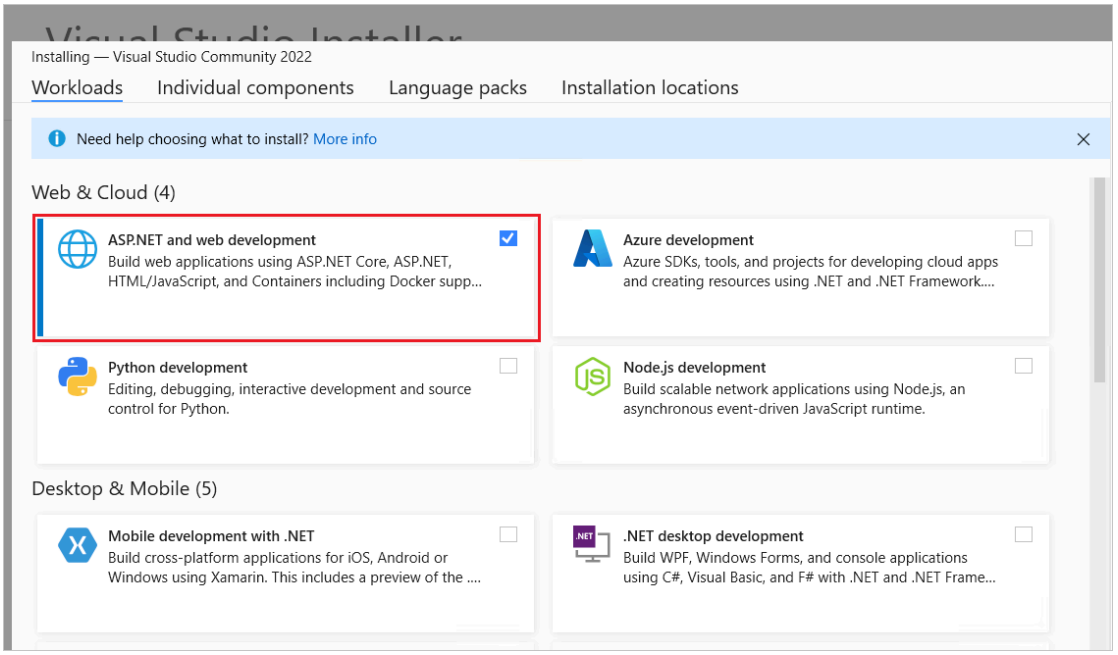
- ✓ Configure MongoDB
- ✓ Create a MongoDB database
- ✓ Define a MongoDB collection and schema
- ✓ Perform MongoDB CRUD operations from a web API
- ✓ Customize JSON serialization

Prerequisites

- MongoDB 6.0.5 or later
- MongoDB Shell

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Configure MongoDB

Enable MongoDB and MongoDB Shell access from anywhere on the development machine (Windows/Linux/macOS):

1. Download and Install MongoDB Shell:

    - macOS/Linux: Choose a directory to extract the MongoDB Shell to. Add the resulting path for `mongosh` to the `PATH` environment variable.
    - Windows: MongoDB Shell (mongosh.exe) is installed at *C:\Users<user>\AppData\Local\Programs\mongosh*. Add the resulting path for `mongosh.exe` to the `PATH` environment variable.

2. Download and Install MongoDB:

    - macOS/Linux: Verify the directory that MongoDB was installed at, usually in */usr/local/mongodb*. Add the resulting path for `mongodb` to the `PATH` environment variable.
    - Windows: MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `PATH` environment variable.

3. Choose a Data Storage Directory: Select a directory on your development machine for storing data. Create the directory if it doesn't exist. The MongoDB Shell doesn't create new directories:

    - macOS/Linux: For example, `/usr/local/var/mongodb`.

- Windows: For example, `C:\\BooksData`.

4. In the OS command shell (not the MongoDB Shell), use the following command to connect to MongoDB on default port 27017. Replace `<data_directory_path>` with the directory chosen in the previous step.

```
Console

mongod --dbpath <data_directory_path>
```

Use the previously installed MongoDB Shell in the following steps to create a database, make collections, and store documents. For more information on MongoDB Shell commands, see mongosh.

1. Open a MongoDB command shell instance by launching `mongosh.exe`.

2. In the command shell, connect to the default test database by running the following command:

```
Console

mongosh
```

3. Run the following command in the command shell:

```
Console

use BookStore
```

   A database named *BookStore* is created if it doesn't already exist. If the database does exist, its connection is opened for transactions.

4. Create a `Books` collection using following command:

```
Console

db.createCollection('Books')
```

   The following result is displayed:

```
Console

{ "ok" : 1 }
```

5. Define a schema for the `Books` collection and insert two documents using the following command:

```
Console

db.Books.insertMany([{ "Name": "Design Patterns", "Price": 54.93,
"Category": "Computers", "Author": "Ralph Johnson" }, { "Name": "Clean
Code", "Price": 43.15, "Category": "Computers","Author": "Robert C.
Martin" }])
```

A result similar to the following is displayed:

```
Console

{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("61a6058e6c43f32854e51f51"),
        ObjectId("61a6058e6c43f32854e51f52")
    ]
}
```

> **Note**
>
> The `ObjectId`s shown in the preceding result won't match those shown in the command shell.

6. View the documents in the database using the following command:

```
Console

db.Books.find().pretty()
```

A result similar to the following is displayed:

```
Console

{
    "_id" : ObjectId("61a6058e6c43f32854e51f51"),
    "Name" : "Design Patterns",
    "Price" : 54.93,
    "Category" : "Computers",
    "Author" : "Ralph Johnson"
}
{
    "_id" : ObjectId("61a6058e6c43f32854e51f52"),
```

```
    "Name" : "Clean Code",
    "Price" : 43.15,
    "Category" : "Computers",
    "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

Create the ASP.NET Core web API project

Visual Studio

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web API** project type, and select **Next**.
3. Name the project *BookStoreApi*, and select **Next**.
4. Select the **.NET 8.0 (Long Term support)** framework and select **Create**.
5. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

PowerShell

```
Install-Package MongoDB.Driver
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

C#

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BookStoreApi.Models;

public class Book
{
```

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }

    [BsonElement("Name")]
    public string BookName { get; set; } = null!;

    public decimal Price { get; set; }

    public string Category { get; set; } = null!;

    public string Author { get; set; } = null!;
}
```

In the preceding class, the `Id` property is:

- Required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Annotated with [BsonId] to make this property the document's primary key.
- Annotated with [BsonRepresentation(BsonType.ObjectId)] to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the [BsonElement] attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to `appsettings.json`:

JSON

```
{
    "BookStoreDatabase": {
        "ConnectionString": "mongodb://localhost:27017",
        "DatabaseName": "BookStore",
        "BooksCollectionName": "Books"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*"
}
```

2. Add a `BookStoreDatabaseSettings` class to the *Models* directory with the following code:

```C#
namespace BookStoreApi.Models;

public class BookStoreDatabaseSettings
{
    public string ConnectionString { get; set; } = null!;

    public string DatabaseName { get; set; } = null!;

    public string BooksCollectionName { get; set; } = null!;
}
```

   The preceding `BookStoreDatabaseSettings` class is used to store the `appsettings.json` file's `BookStoreDatabase` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));
```

   In the preceding code, the configuration instance to which the `appsettings.json` file's `BookStoreDatabase` section binds is registered in the Dependency Injection (DI) container. For example, the `BookStoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookStoreDatabase:ConnectionString` property in `appsettings.json`.

4. Add the following code to the top of `Program.cs` to resolve the `BookStoreDatabaseSettings` reference:

```C#
using BookStoreApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.

2. Add a `BooksService` class to the *Services* directory with the following code:

C#

```
using BookStoreApi.Models;
using Microsoft.Extensions.Options;
using MongoDB.Driver;

namespace BookStoreApi.Services;

public class BooksService
{
    private readonly IMongoCollection<Book> _booksCollection;

    public BooksService(
        IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
    {
        var mongoClient = new MongoClient(
            bookStoreDatabaseSettings.Value.ConnectionString);

        var mongoDatabase = mongoClient.GetDatabase(
            bookStoreDatabaseSettings.Value.DatabaseName);

        _booksCollection = mongoDatabase.GetCollection<Book>(
            bookStoreDatabaseSettings.Value.BooksCollectionName);
    }

    public async Task<List<Book>> GetAsync() =>
        await _booksCollection.Find(_ => true).ToListAsync();

    public async Task<Book?> GetAsync(string id) =>
        await _booksCollection.Find(x => x.Id ==
id).FirstOrDefaultAsync();

    public async Task CreateAsync(Book newBook) =>
        await _booksCollection.InsertOneAsync(newBook);

    public async Task UpdateAsync(string id, Book updatedBook) =>
        await _booksCollection.ReplaceOneAsync(x => x.Id == id,
updatedBook);

    public async Task RemoveAsync(string id) =>
        await _booksCollection.DeleteOneAsync(x => x.Id == id);
}
```

In the preceding code, a `BookStoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the `appsettings.json` configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));

builder.Services.AddSingleton<BooksService>();
```

In the preceding code, the `BooksService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BooksService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of `Program.cs` to resolve the `BooksService` reference:

```C#
using BookStoreApi.Services;
```

The `BooksService` class uses the following `MongoDB.Driver` members to run CRUD operations against the database:

- MongoClient: Reads the server instance for running database operations. The constructor of this class is provided in the MongoDB connection string:

```C#
public BooksService(
    IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
{
    var mongoClient = new MongoClient(
        bookStoreDatabaseSettings.Value.ConnectionString);

    var mongoDatabase = mongoClient.GetDatabase(
        bookStoreDatabaseSettings.Value.DatabaseName);

    _booksCollection = mongoDatabase.GetCollection<Book>(
        bookStoreDatabaseSettings.Value.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for running operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Run CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOneAsync: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOneAsync: Inserts the provided object as a new document in the collection.
- ReplaceOneAsync: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

C#

```
using BookStoreApi.Models;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private readonly BooksService _booksService;

    public BooksController(BooksService booksService) =>
        _booksService = booksService;

    [HttpGet]
    public async Task<List<Book>> Get() =>
        await _booksService.GetAsync();

    [HttpGet("{id:length(24)}")]
```

```
    public async Task<ActionResult<Book>> Get(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        return book;
    }

    [HttpPost]
    public async Task<IActionResult> Post(Book newBook)
    {
        await _booksService.CreateAsync(newBook);

        return CreatedAtAction(nameof(Get), new { id = newBook.Id },
newBook);
    }

    [HttpPut("{id:length(24)}")]
    public async Task<IActionResult> Update(string id, Book updatedBook)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        updatedBook.Id = book.Id;

        await _booksService.UpdateAsync(id, updatedBook);

        return NoContent();
    }

    [HttpDelete("{id:length(24)}")]
    public async Task<IActionResult> Delete(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        await _booksService.RemoveAsync(id);

        return NoContent();
    }
}
```

The preceding web API controller:

- Uses the `BooksService` class to run CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtAction in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtAction` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.

2. Navigate to `https://localhost:<port>/api/books`, where `<port>` is the automatically assigned port number for the app, to test the controller's parameterless `Get` action method, select **Try it out** > **Execute**. A JSON response similar to the following is displayed:

   JSON

   ```
   [
     {
       "id": "61a6058e6c43f32854e51f51",
       "bookName": "Design Patterns",
       "price": 54.93,
       "category": "Computers",
       "author": "Ralph Johnson"
     },
     {
       "id": "61a6058e6c43f32854e51f52",
       "bookName": "Clean Code",
       "price": 43.15,
       "category": "Computers",
       "author": "Robert C. Martin"
     }
   ]
   ```

3. Navigate to `https://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. A JSON response similar to the following is displayed:

   JSON

   ```
   {
     "id": "61a6058e6c43f32854e51f52",
   ```

```
  "bookName": "Clean Code",
  "price": 43.15,
  "category": "Computers",
  "author": "Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Program.cs`, chain the following highlighted code on to the `AddControllers` method call:

   C#

   ```
   var builder = WebApplication.CreateBuilder(args);

   // Add services to the container.
   builder.Services.Configure<BookStoreDatabaseSettings>(
       builder.Configuration.GetSection("BookStoreDatabase"));

   builder.Services.AddSingleton<BooksService>();

   builder.Services.AddControllers()
       .AddJsonOptions(
           options => options.JsonSerializerOptions.PropertyNamingPolicy =
   null);
   ```

   With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author` instead of `author`.

2. In `Models/Book.cs`, annotate the `BookName` property with the [JsonPropertyName] attribute:

   C#

   ```
   [BsonElement("Name")]
   [JsonPropertyName("Name")]
   ```

```
public string BookName { get; set; } = null!;
```

The `[JsonPropertyName]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of `Models/Book.cs` to resolve the `[JsonProperty]` attribute reference:

```
C#

using System.Text.Json.Serialization;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

Additional resources

- View or download sample code (how to download)
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Create a web API with ASP.NET Core

Tutorial: Call an ASP.NET Core web API with JavaScript

Article • 04/25/2024

By Rick Anderson

This tutorial shows how to call an ASP.NET Core web API with JavaScript, using the Fetch API.

Prerequisites

- Complete Tutorial: Create a web API
- Familiarity with CSS, HTML, and JavaScript

Call the web API with JavaScript

In this section, you'll add an HTML page containing forms for creating and managing to-do items. Event handlers are attached to elements on the page. The event handlers result in HTTP requests to the web API's action methods. The Fetch API's `fetch` function initiates each HTTP request.

The `fetch` function returns a Promise object, which contains an HTTP response represented as a `Response` object. A common pattern is to extract the JSON response body by invoking the `json` function on the `Response` object. JavaScript updates the page with the details from the web API's response.

The simplest `fetch` call accepts a single parameter representing the route. A second parameter, known as the `init` object, is optional. `init` is used to configure the HTTP request.

1. Configure the app to serve static files and enable default file mapping. The following highlighted code is needed in `Program.cs`:

C#

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
```

```
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));

var app = builder.Build();

if (builder.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseDefaultFiles();
app.UseStaticFiles();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

1. Create a *wwwroot* folder in the project root.
2. Create a *css* folder inside of the *wwwroot* folder.
3. Create a *js* folder inside of the *wwwroot* folder.
4. Add an HTML file named `index.html` to the *wwwroot* folder. Replace the contents of `index.html` with the following markup:

HTML

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <link rel="stylesheet" href="css/site.css" />
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST"
onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="editForm">
        <h3>Edit</h3>
        <form action="javascript:void(0);" onsubmit="updateItem()">
            <input type="hidden" id="edit-id">
```

```
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete?</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="js/site.js" asp-append-version="true"></script>
    <script type="text/javascript">
        getItems();
    </script>
</body>
</html>
```

5. Add a CSS file named `site.css` to the *wwwroot/css* folder. Replace the contents of `site.css` with the following styles:

```css
input[type='submit'], button, [aria-label] {
    cursor: pointer;
}

#editForm {
    display: none;
}

table {
    font-family: Arial, sans-serif;
    border: 1px solid;
    border-collapse: collapse;
}

th {
    background-color: #f8f8f8;
    padding: 5px;
}

td {
    border: 1px solid;
```

```
    padding: 5px;
}
```

6. Add a JavaScript file named `site.js` to the *wwwroot/js* folder. Replace the contents of `site.js` with the following code:

JavaScript

```
const uri = 'api/todoitems';
let todos = [];

function getItems() {
  fetch(uri)
    .then(response => response.json())
    .then(data => _displayItems(data))
    .catch(error => console.error('Unable to get items.', error));
}

function addItem() {
  const addNameTextbox = document.getElementById('add-name');

  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}

function deleteItem(id) {
  fetch(`${uri}/${id}`, {
    method: 'DELETE'
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to delete item.', error));
}

function displayEditForm(id) {
  const item = todos.find(item => item.id === id);
```

```
  document.getElementById('edit-name').value = item.name;
  document.getElementById('edit-id').value = item.id;
  document.getElementById('edit-isComplete').checked = item.isComplete;
  document.getElementById('editForm').style.display = 'block';
}

function updateItem() {
  const itemId = document.getElementById('edit-id').value;
  const item = {
    id: parseInt(itemId, 10),
    isComplete: document.getElementById('edit-isComplete').checked,
    name: document.getElementById('edit-name').value.trim()
  };

  fetch(`${uri}/${itemId}`, {
    method: 'PUT',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to update item.', error));

  closeInput();

  return false;
}

function closeInput() {
  document.getElementById('editForm').style.display = 'none';
}

function _displayCount(itemCount) {
  const name = (itemCount === 1) ? 'to-do' : 'to-dos';

  document.getElementById('counter').innerText = `${itemCount}
${name}`;
}

function _displayItems(data) {
  const tBody = document.getElementById('todos');
  tBody.innerHTML = '';

  _displayCount(data.length);

  const button = document.createElement('button');

  data.forEach(item => {
    let isCompleteCheckbox = document.createElement('input');
    isCompleteCheckbox.type = 'checkbox';
    isCompleteCheckbox.disabled = true;
    isCompleteCheckbox.checked = item.isComplete;
```

```
    let editButton = button.cloneNode(false);
    editButton.innerText = 'Edit';
    editButton.setAttribute('onclick', `displayEditForm(${item.id})`);

    let deleteButton = button.cloneNode(false);
    deleteButton.innerText = 'Delete';
    deleteButton.setAttribute('onclick', `deleteItem(${item.id})`);

    let tr = tBody.insertRow();

    let td1 = tr.insertCell(0);
    td1.appendChild(isCompleteCheckbox);

    let td2 = tr.insertCell(1);
    let textNode = document.createTextNode(item.name);
    td2.appendChild(textNode);

    let td3 = tr.insertCell(2);
    td3.appendChild(editButton);

    let td4 = tr.insertCell(3);
    td4.appendChild(deleteButton);
  });

  todos = data;
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

1. Open *Properties\launchSettings.json*.
2. Remove the `launchUrl` property to force the app to open at `index.html`—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the web API requests.

Get a list of to-do items

In the following code, an HTTP GET request is sent to the *api/todoitems* route:

JavaScript

```
fetch(uri)
  .then(response => response.json())
  .then(data => _displayItems(data))
  .catch(error => console.error('Unable to get items.', error));
```

When the web API returns a successful status code, the `_displayItems` function is invoked. Each to-do item in the array parameter accepted by `_displayItems` is added to a table with **Edit** and **Delete** buttons. If the web API request fails, an error is logged to the browser's console.

Add a to-do item

In the following code:

- An `item` variable is declared to construct an object literal representation of the to-do item.
- A Fetch request is configured with the following options:
  - `method`—specifies the POST HTTP action verb.
  - `body`—specifies the JSON representation of the request body. The JSON is produced by passing the object literal stored in `item` to the JSON.stringify function.
  - `headers`—specifies the `Accept` and `Content-Type` HTTP request headers. Both headers are set to `application/json` to specify the media type being received and sent, respectively.
- An HTTP POST request is sent to the *api/todoitems* route.

JavaScript

```
function addItem() {
  const addNameTextbox = document.getElementById('add-name');

  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}
```

When the web API returns a successful status code, the `getItems` function is invoked to update the HTML table. If the web API request fails, an error is logged to the browser's console.

Update a to-do item

Updating a to-do item is similar to adding one; however, there are two significant differences:

- The route is suffixed with the unique identifier of the item to update. For example, *api/todoitems/1*.
- The HTTP action verb is PUT, as indicated by the `method` option.

JavaScript

```
fetch(`${uri}/${itemId}`, {
  method: 'PUT',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(item)
})
.then(() => getItems())
.catch(error => console.error('Unable to update item.', error));
```

Delete a to-do item

To delete a to-do item, set the request's `method` option to `DELETE` and specify the item's unique identifier in the URL.

JavaScript

```
fetch(`${uri}/${id}`, {
  method: 'DELETE'
})
.then(() => getItems())
.catch(error => console.error('Unable to delete item.', error));
```

Advance to the next tutorial to learn how to generate web API help pages:

Get started with Swashbuckle and ASP.NET Core

Create backend services for native mobile apps with ASP.NET Core

Article • 07/23/2024

By James Montemagno

Mobile apps can communicate with ASP.NET Core backend services. For instructions on connecting local web services from iOS simulators and Android emulators, see Connect to Local Web Services from iOS Simulators and Android Emulators.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core to support native mobile apps. It uses the Xamarin.Forms TodoRest app as its native client, which includes separate native clients for Android, iOS, and Windows. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), and download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Core Web API services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

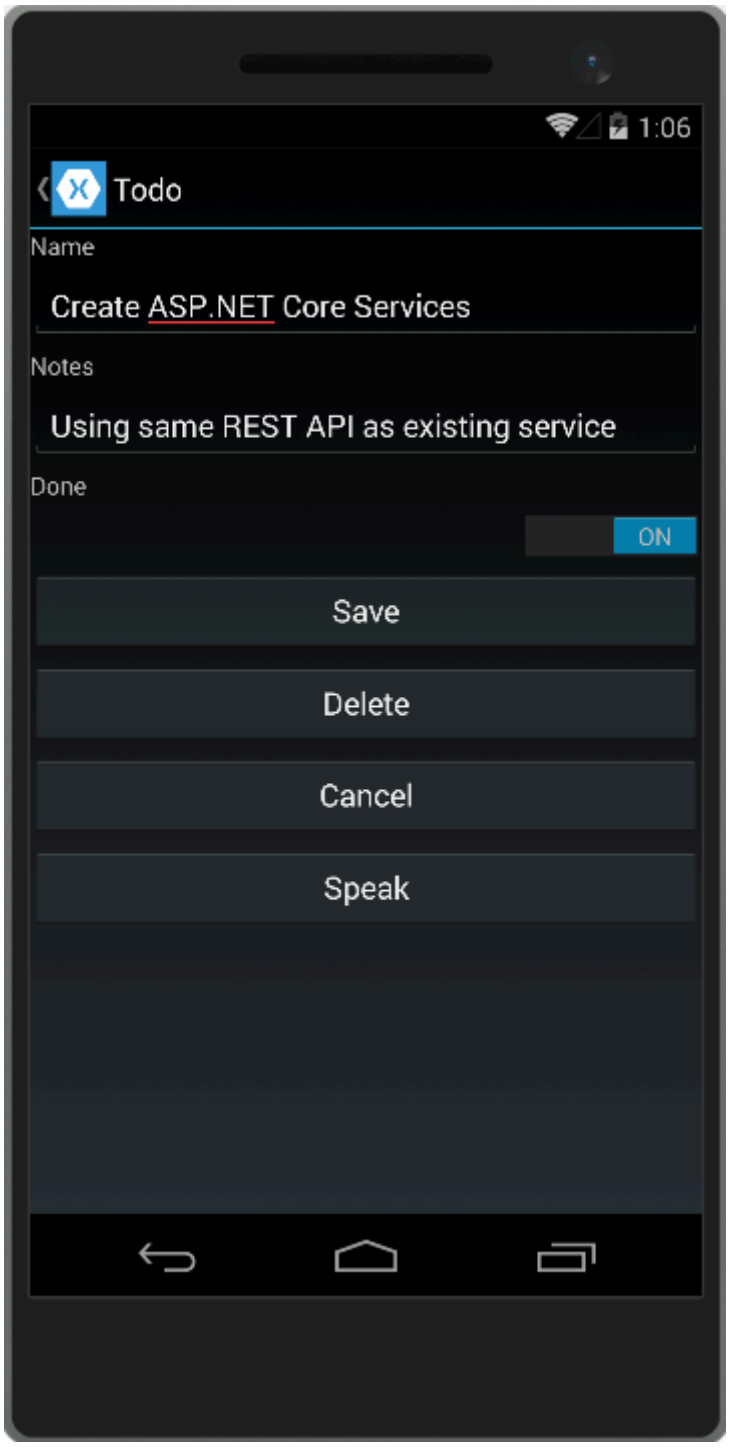
Features

The TodoREST app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

In the previous example, The main view of the items lists each item's name and indicates if it's done with a checkmark.

Tapping the `+` icon opens an add item dialog:

Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



To test it out yourself against the ASP.NET Core app created in the next section running on your computer, update the app's RestUrl constant.

Android emulators don't run on the local machine and use a loopback IP (10.0.2.2) to communicate with the local machine. Use Xamarin.Essentials DeviceInfo to detect what operating the system is running to use the correct URL.

Navigate to the TodoREST project and open the Constants.cs file. The `Constants.cs` file contains the following configuration.

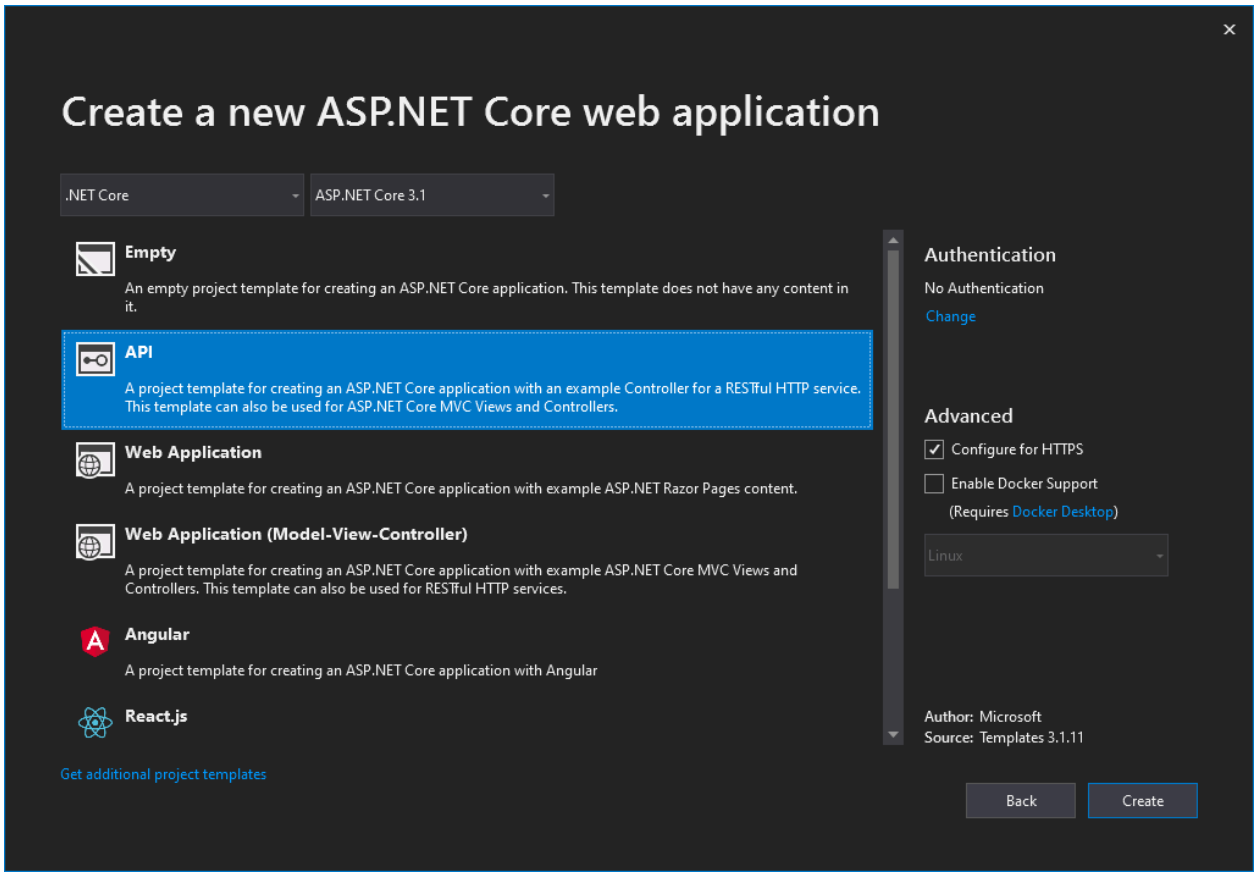
```
C#
```

```
using Xamarin.Essentials;
using Xamarin.Forms;

namespace TodoREST
{
    public static class Constants
    {
        // URL of REST service
        //public static string RestUrl =
"https://YOURPROJECT.azurewebsites.net:8081/api/todoitems/{0}";

        // URL of REST service (Android does not use localhost)
        // Use http cleartext for local deployment. Change to https for
production
        public static string RestUrl = DeviceInfo.Platform ==
DevicePlatform.Android ? "http://10.0.2.2:5000/api/todoitems/{0}" :
"http://localhost:5000/api/todoitems/{0}";
    }
}
```

You can optionally deploy the web service to a cloud service such as Azure and update the `RestUrl`.

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template. Name the project *TodoAPI*.



The app should respond to all requests made to port 5000 including clear-text HTTP traffic for our mobile client. Update `Startup.cs` so UseHttpsRedirection doesn't run in development:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // For mobile apps, allow http traffic.
        app.UseHttpsRedirection();
    }

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

> **Note**
>
> Run the app directly, rather than behind IIS Express. IIS Express ignores non-local requests by default. Run **dotnet run** from a command prompt, or choose the app name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields with the `[Required]` attribute:

C#

```
using System.ComponentModel.DataAnnotations;

namespace TodoAPI.Models
{
    public class TodoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}
```

The API methods require some way to work with data. Use the same `ITodoRepository` interface the original Xamarin sample uses:

C#

```
using System.Collections.Generic;
using TodoAPI.Models;

namespace TodoAPI.Interfaces
{
    public interface ITodoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<TodoItem> All { get; }
        TodoItem Find(string id);
        void Insert(TodoItem item);
        void Update(TodoItem item);
        void Delete(string id);
```

```
    }
}
```

For this sample, the implementation just uses a private collection of items:

C#

```
using System.Collections.Generic;
using System.Linq;
using TodoAPI.Interfaces;
using TodoAPI.Models;

namespace TodoAPI.Services
{
    public class TodoRepository : ITodoRepository
    {
        private List<TodoItem> _todoList;

        public TodoRepository()
        {
            InitializeData();
        }

        public IEnumerable<TodoItem> All
        {
            get { return _todoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _todoList.Any(item => item.ID == id);
        }

        public TodoItem Find(string id)
        {
            return _todoList.FirstOrDefault(item => item.ID == id);
        }

        public void Insert(TodoItem item)
        {
            _todoList.Add(item);
        }

        public void Update(TodoItem item)
        {
            var todoItem = this.Find(item.ID);
            var index = _todoList.IndexOf(todoItem);
            _todoList.RemoveAt(index);
            _todoList.Insert(index, item);
        }

        public void Delete(string id)
        {
```

```
            _todoList.Remove(this.Find(id));
        }

        private void InitializeData()
        {
            _todoList = new List<TodoItem>();

            var todoItem1 = new TodoItem
            {
                ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
                Name = "Learn app development",
                Notes = "Take Microsoft Learn Courses",
                Done = true
            };

            var todoItem2 = new TodoItem
            {
                ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
                Name = "Develop apps",
                Notes = "Use Visual Studio and Visual Studio for Mac",
                Done = false
            };

            var todoItem3 = new TodoItem
            {
                ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
                Name = "Publish apps",
                Notes = "All app stores",
                Done = false,
            };

            _todoList.Add(todoItem1);
            _todoList.Add(todoItem2);
            _todoList.Add(todoItem3);
        }
    }
}
```

Configure the implementation in `Startup.cs`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<ITodoRepository, TodoRepository>();
    services.AddControllers();
}
```

Creating the Controller

Add a new controller to the project, TodoItemsController. It should inherit from ControllerBase. Add a `Route` attribute to indicate that the controller handles requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `ITodoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance is provided using the framework's support for dependency injection.

C#

```
[ApiController]
[Route("api/[controller]")]
public class TodoItemsController : ControllerBase
{
    private readonly ITodoRepository _todoRepository;

    public TodoItemsController(ITodoRepository todoRepository)
    {
        _todoRepository = todoRepository;
    }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP `GET` request.

Test the API using curl

You can test the API method using a variety of tools. For this tutorial the following open source command-line tools are used:

- curl: Transfers data using various protocols including HTTP and HTTPS. curl is used in this tutorial to call the API using HTTP methods `GET`, `POST`, `PUT`, and `DELETE`.
- jq: A JSON processor used in this tutorial to format JSON data so that it's easy to read from the API response.

Install curl and jq

Windows

curl is installed with Windows 10, version 1802 or higher. For more information on installing curl, see the Official curl website.

Install jq with the following command in PowerShell or the Command Prompt:

PowerShell

```
winget install jqlang.jq
```

The `jq` command is available once the PowerShell or Command Prompt is closed and restarted.

For more details on jq installation, see jq.

Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

C#

```
[HttpGet]
public IActionResult List()
{
    return Ok(_todoRepository.All);
}
```

Windows

> **Note**
>
> Windows PowerShell 5.1 recognizes `curl` as an alias to `Invoke-WebRequest`. To use `curl.exe` instead, type the `&` operator followed by the full path to `curl.exe`. Find the full path to `curl.exe` by typing `where curl` in the Command Prompt. For example, if the full path to `curl.exe` is `C:\Windows\System32\curl.exe`. Then instead of typing the command `curl --`

`help`, use `& 'C:\Windows\System32\curl.exe' --help`. PowerShell 7 uses `curl` as the command for `curl.exe` and so a full path is not required.

In PowerShell, call the following curl command:

PowerShell

```
curl -v -X GET 'http://localhost:5000/api/todoitems/' | jq
```

The previous curl command includes the following components:

- `-v`: Activates verbose mode, providing detailed information about the HTTP response and is useful for API testing and troubleshooting.
- `-X GET`: Specifies the use of the HTTP `GET` method for the request. While curl can often infer the intended HTTP method, this option makes it explicit.
- `'http://localhost:5000/api/todoitems/'`: This is the request's target URL. In this instance, it's a REST API endpoint.
- `| jq`: This segment isn't related to curl directly. The pipe `|` is a shell operator that takes the output from the command on its left and "pipes" it to the command on its right. `jq` is a command-line JSON processor. While not required, `jq` makes the returned JSON data easier to read.

The `List` method returns a 200 OK response code and all of the Todo items, serialized as JSON:

Output


```
[
  {
    "id": "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
    "name": "Learn app development",
    "notes": "Take Microsoft Learn Courses",
    "done": true
  },
  {
    "id": "b94afb54-a1cb-4313-8af3-b7511551b33b",
    "name": "Develop apps",
    "notes": "Use Visual Studio and Visual Studio for Mac",
    "done": false
  },
  {
    "id": "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
    "name": "Publish apps",
    "notes": "All app stores",
    "done": false
```

```
  }
]
```

Creating Items

By convention, creating new data items is mapped to the HTTP `POST` verb. The `Create` method has an `[HttpPost]` attribute applied to it and accepts a `TodoItem` instance. Since the `item` argument is passed in the body of the POST, this parameter specifies the `[FromBody]` attribute.

Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it's added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

C#

```
[HttpPost]
public IActionResult Create([FromBody]TodoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return
BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _todoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict,
ErrorCode.TodoItemIDInUse.ToString());
        }
        _todoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

The sample uses an `enum` containing error codes that are passed to the mobile client:

C#

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

In the terminal, test adding new items by calling the following curl command using the `POST` verb and providing the new object in JSON format in the Body of the request.

Windows

PowerShell

```
curl -v -X POST 'http://localhost:5000/api/todoitems/' `
--header 'Content-Type: application/json' `
--data '{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87e243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": false
}' | jq
```

The previous curl command includes the following options:

- `--header 'Content-Type: application/json'`: Sets the `Content-Type` header to `application/json`, indicating that the request body contains JSON data.
- `--data '{...}'`: Sends the specified data in the request body.

The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP `PUT` requests. Other than this change, the `Edit` method is almost identical to `Create`. If the record isn't found, the `Edit` action returns a `NotFound` (404) response.

C#

```
[HttpPut]
public IActionResult Edit([FromBody] TodoItem item)
```

```
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return
BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _todoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with curl, change the verb to `PUT`. Specify the updated object data in the Body of the request.

Windows

PowerShell

```
curl -v -X PUT 'http://localhost:5000/api/todoitems/' `
--header 'Content-Type: application/json' `
--data '{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87e243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": true
}' | jq
```

This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making `DELETE` requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist receive `NotFound` responses. Otherwise, a successful request returns a `NoContent` (204) response.

C#

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Test with curl by changing the HTTP verb to `DELETE` and appending the ID of the data object to delete at the end of the URL. Nothing is required in the Body of the request.

Windows

PowerShell

```
curl -v -X DELETE 'http://localhost:5000/api/todoitems/6bb8b868-dba1-4f1a-93b7-24ebce87e243'
```

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.

- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, see Prevent over-posting

Common Web API Conventions

As you develop the backend services for your app, you'll want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown previously, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Additional resources

- Xamarin.Forms: Web Service Authentication
- Xamarin.Forms: Consume a RESTful Web Service
- Consume REST web services in Xamarin Apps
- Create a web API with ASP.NET Core

Publish an ASP.NET Core web API to Azure API Management with Visual Studio

Article • 06/18/2024

By Matt Soucoup

In this tutorial you'll learn how to create an ASP.NET Core web API project using Visual Studio, ensure it has OpenAPI support, and then publish the web API to both Azure App Service and Azure API Management.

Set up

To complete the tutorial you'll need an Azure account.

- Open a free Azure account if you don't have one.

Create an ASP.NET Core web API

Visual Studio allows you to easily create a new ASP.NET Core web API project from a template. Follow these directions to create a new ASP.NET Core web API project:

- From the File menu, select **New** > **Project**.
- Enter **Web API** in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project **WeatherAPI** and select **Next**.
- In the **Additional information** dialog:
- Confirm the Framework is **.NET 6.0 (Long-term support)**.
- Confirm the checkbox for **Use controllers (uncheck to use minimal APIs)** is checked.
- Confirm the checkbox for **Enable OpenAPI support** is checked.
- Select **Create**.

Explore the code

Swagger definitions allow Azure API Management to read the app's API definitions. By checking the **Enable OpenAPI support** checkbox during app creation, Visual Studio

automatically adds the code to create the Swagger definitions. Open up the `Program.cs` file which shows the following code:

```
C#


...

builder.Services.AddSwaggerGen();

...

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(); // Protected by if (env.IsDevelopment())
}

...
```

Ensure the Swagger definitions are always generated

Azure API Management needs the Swagger definitions to always be present, regardless of the application's environment. To ensure they are always generated, move `app.UseSwagger();` outside of the `if (app.Environment.IsDevelopment())` block.

The updated code:

```
C#


...

app.UseSwagger();

if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI();
}

...
```

Change the API routing

Change the URL structure needed to access the `Get` action of the `WeatherForecastController`. Complete the following steps:

1. Open the `WeatherForecastController.cs` file.
2. Replace the `[Route("[controller]")]` class-level attribute with `[Route("/")]`. The updated class definition :

```
C#

[ApiController]
[Route("/")]
public class WeatherForecastController : ControllerBase
```

Publish the web API to Azure App Service

Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

1. Publish the API app to Azure App Service.
2. Publish the ASP.NET Core web API app to the Azure API Management service instance.

Publish the API app to Azure App Service

Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

1. In **Solution Explorer**, right-click the project and select **Publish**.
2. In the **Publish** dialog, select **Azure** and select the **Next** button.
3. Select **Azure App Service (Windows)** and select the **Next** button.
4. Select **Create a new Azure App Service**.

   The **Create App Service** dialog appears. The **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them.
5. Select the **Create** button.
6. Once the app service is created, select the **Next** button.

7. Select **Create a new API Management Service**.

   The **Create API Management Service** dialog appears. You can leave the **API Name**, **Subscription Name**, and **Resource Group** entry fields as they are. Select the **new** button next to the **API Management Service** entry and enter the required fields from that dialog box.

   Select the **OK** button to create the API Management service.

8. Select the **Create** button to proceed with the API Management service creation. This step may take several minutes to complete.

9. When that completes, select the **Finish** button.

10. The dialog closes and a summary screen appears with information about the publish. Select the **Publish** button.

    The web API publishes to both Azure App Service and Azure API Management. A new browser window will appear and show the API running in Azure App Service. You can close that window.

11. Open up the Azure portal in a web browser and navigate to the API Management instance you created.

12. Select the **APIs** option from the left-hand menu.

13. Select the API you created in the preceding steps. It's now populated and you can explore around.

Configure the published API name

Notice the name of the API is named *WeatherAPI*; however, we would like to call it *Weather Forecasts*. Complete the following steps to update the name:

1. Add the following to `Program.cs` immediately after `servies.AddSwaggerGen();`

```
C#
```

```
builder.Services.ConfigureSwaggerGen(setup =>
{
    setup.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo
    {
        Title = "Weather Forecasts",
        Version = "v1"
    });
});
```

2. Republish the ASP.NET Core web API and open the Azure API Management instance in the Azure portal.

3. Refresh the page in your browser. You'll see the name of the API is now correct.

Verify the web API is working

You can test the deployed ASP.NET Core web API in Azure API Management from the Azure portal with the following steps:

1. Open the **Test** tab.
2. Select **/** or the **Get** operation.
3. Select **Send**.

Clean up

When you've finished testing the app, go to the Azure portal and delete the app.

1. Select **Resource groups**, then select the resource group you created.

2. In the **Resource groups** page, select **Delete**.

3. Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Additional resources

- Azure API Management
- Azure App Service

Tutorial: Create a minimal API with ASP.NET Core

Article • 08/21/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson and Tom Dykstra

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They're ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core.

This tutorial teaches the basics of building a minimal API with ASP.NET Core. Another approach to creating APIs in ASP.NET Core is to use controllers. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating an API project based on controllers that contains more features, see Create a web API.

Overview

This tutorial creates the following API:

Expand table

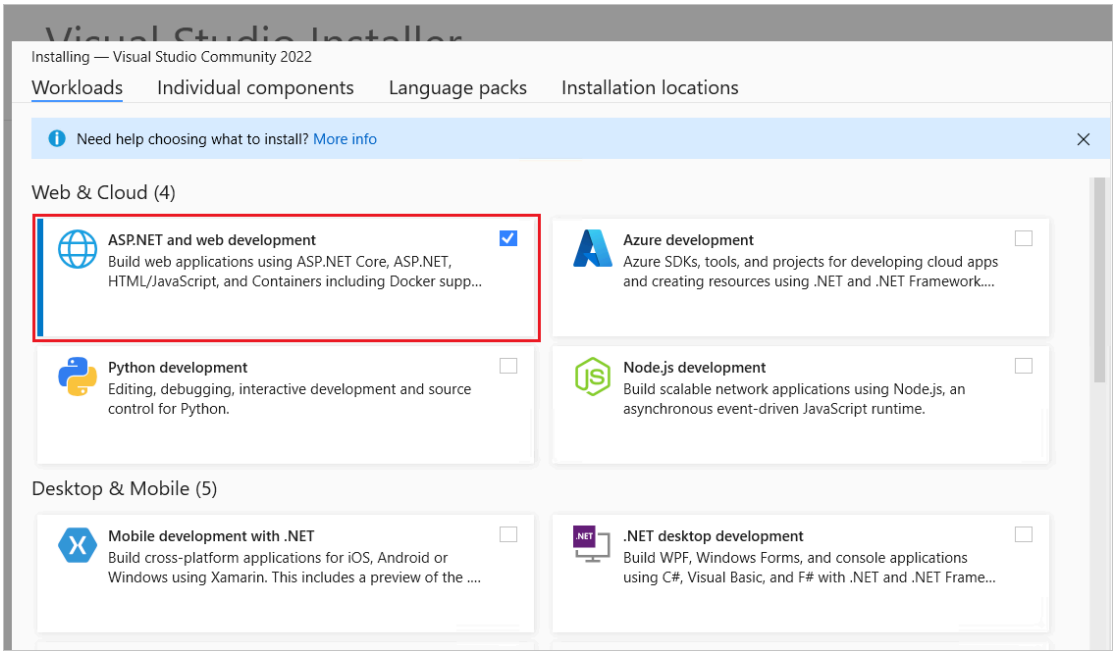
API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item
`POST /todoitems`	Add a new item	To-do item	To-do item
`PUT /todoitems/{id}`	Update an existing item	To-do item	None

API	Description	Request body	Response body
`DELETE /todoitems/{id}`	Delete an item	None	None

Prerequisites

Visual Studio

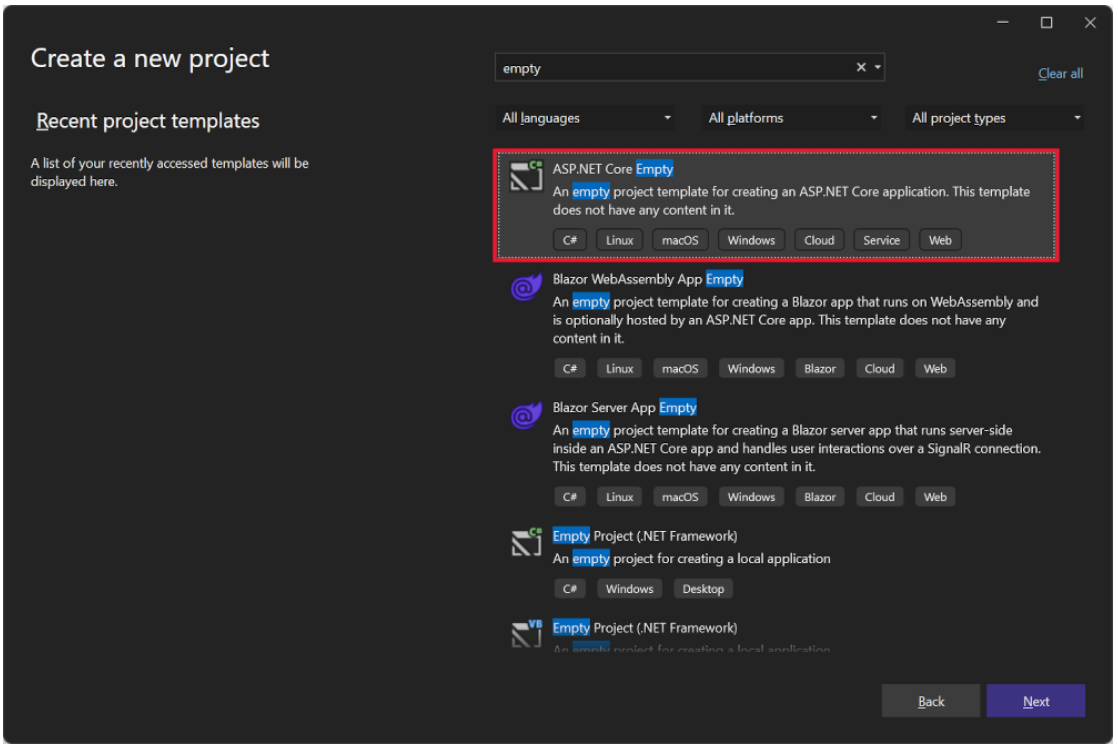
- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



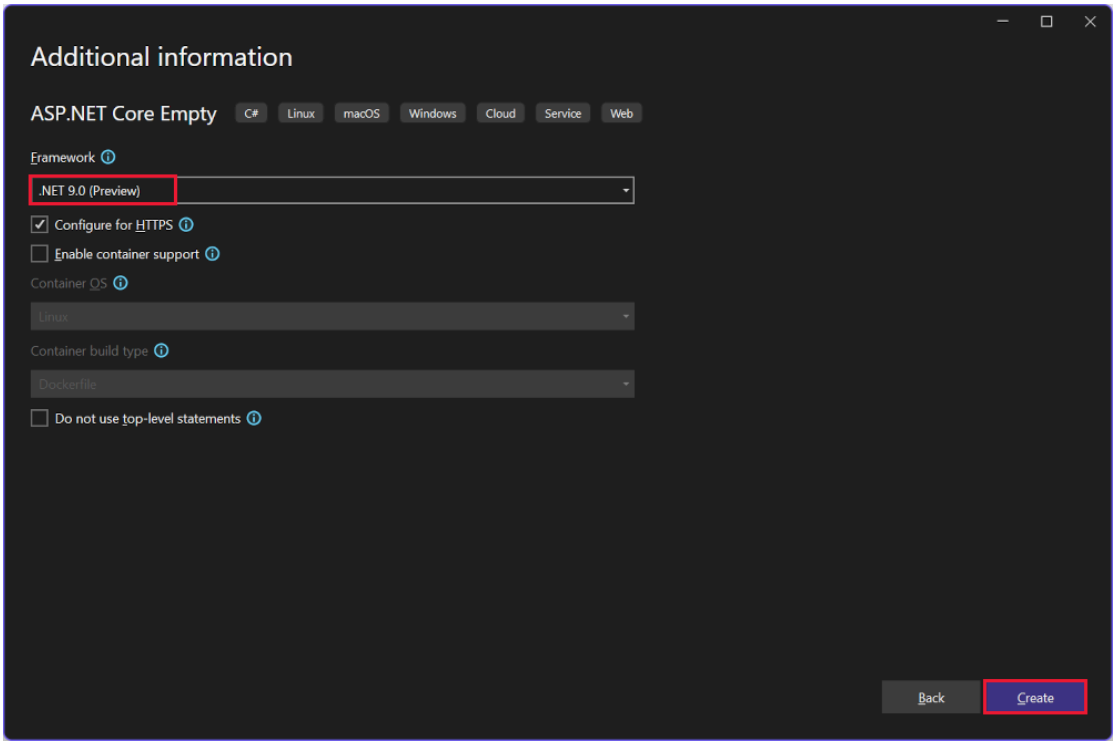
Create an API project

Visual Studio

- Start Visual Studio 2022 and select **Create a new project**.
- In the **Create a new project** dialog:
  - Enter `Empty` in the **Search for templates** search box.
  - Select the **ASP.NET Core Empty** template and select **Next**.



- Name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 9.0 (Preview)**
  - Uncheck **Do not use top-level statements**
  - Select **Create**



Examine the code

The `Program.cs` file contains the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code:

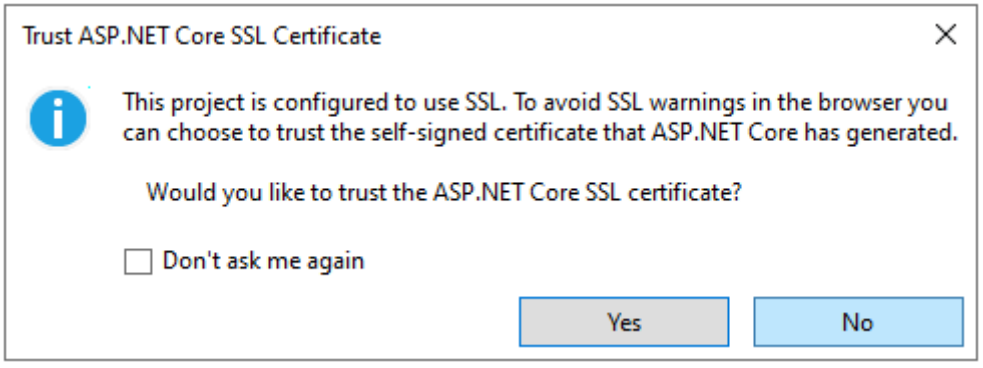
- Creates a WebApplicationBuilder and a WebApplication with preconfigured defaults.
- Creates an HTTP GET endpoint `/` that returns `Hello World!`:

Run the app

Visual Studio

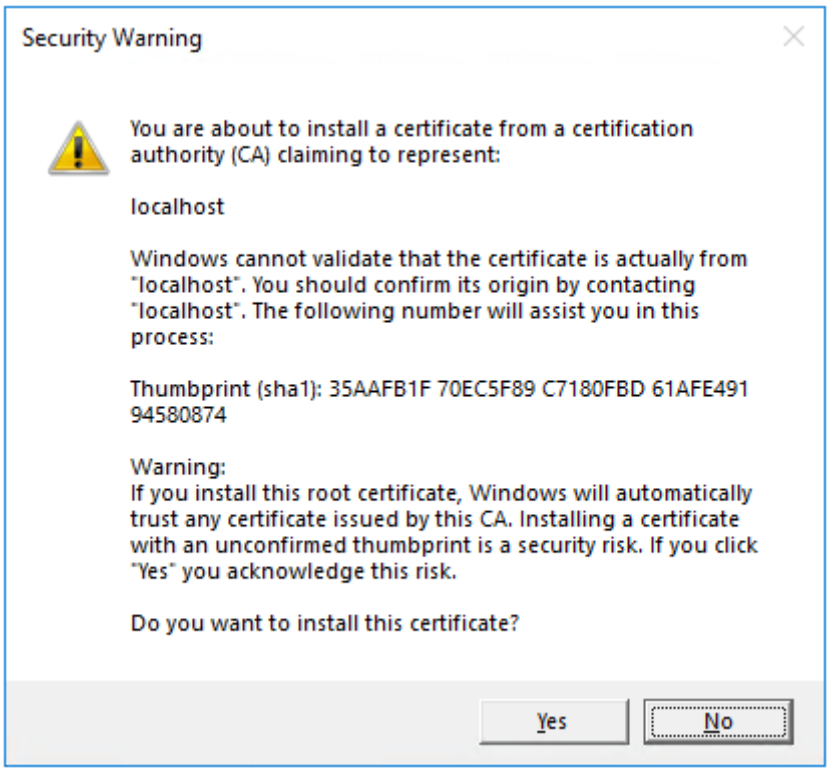
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the Kestrel web server and opens a browser window.

`Hello World!` is displayed in the browser. The `Program.cs` file contains a minimal but complete app.

Close the browser window.

Add NuGet packages

NuGet packages must be added to support the database and diagnostics used in this tutorial.

Visual Studio

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Select **Include Prelease**.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.

- Select the **Project** checkbox in the right pane and then select **Install**.
- Follow the preceding instructions to add the `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` package.

The model and database context classes

- In the project folder, create a file named `Todo.cs` with the following code:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The preceding code creates the model for this app. A *model* is a class that represents data that the app manages.

- Create a file named `TodoDb.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

class TodoDb : DbContext
{
    public TodoDb(DbContextOptions<TodoDb> options)
        : base(options) { }

    public DbSet<Todo> Todos => Set<Todo>();
}
```

The preceding code defines the *database context*, which is the main class that coordinates Entity Framework functionality for a data model. This class derives from the Microsoft.EntityFrameworkCore.DbContext class.

Add the API code

- Replace the contents of the `Program.cs` file with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

```
app.Run();
```

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();
```

The DI container provides access to the database context and other services.

Visual Studio

This tutorial uses Endpoints Explorer and .http files to test the API.

Test posting data

The following code in `Program.cs` creates an HTTP POST endpoint `/todoitems` that adds data to the in-memory database:

C#

```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});
```

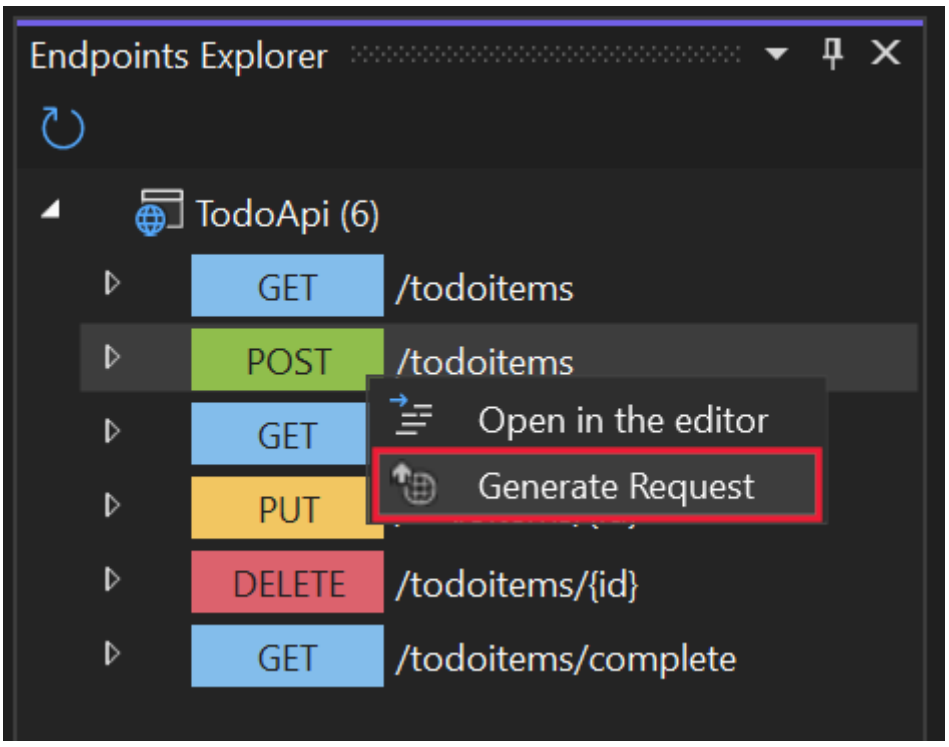
Run the app. The browser displays a 404 error because there's no longer a `/` endpoint.

The POST endpoint will be used to add data to the app.

Visual Studio

- Select **View** > **Other Windows** > **Endpoints Explorer**.

- Right-click the **POST** endpoint and select **Generate request**.



A new file is created in the project folder named `TodoApi.http`, with contents similar to the following example:

```
@TodoApi_HostAddress = https://localhost:7031

Post {{TodoApi_HostAddress}}/todoitems

###
```

  - The first line creates a variable that is used for all of the endpoints.
  - The next line defines a POST request.
  - The triple hashtag (`###`) line is a request delimiter: what comes after it is for a different request.

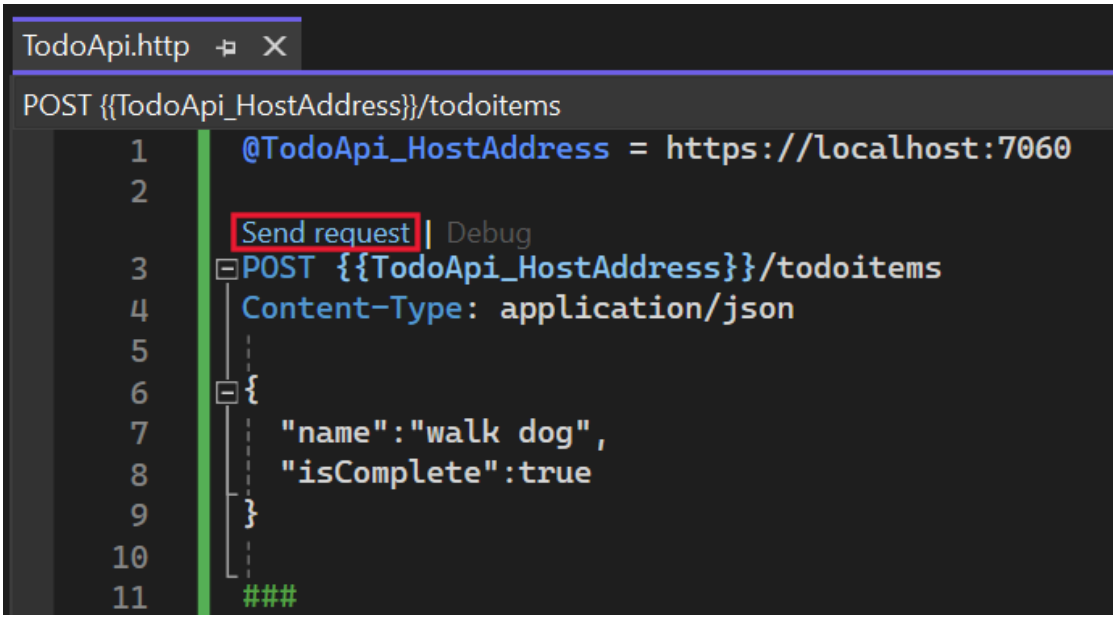
- The POST request needs headers and a body. To define those parts of the request, add the following lines immediately after the POST request line:

```
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}
```

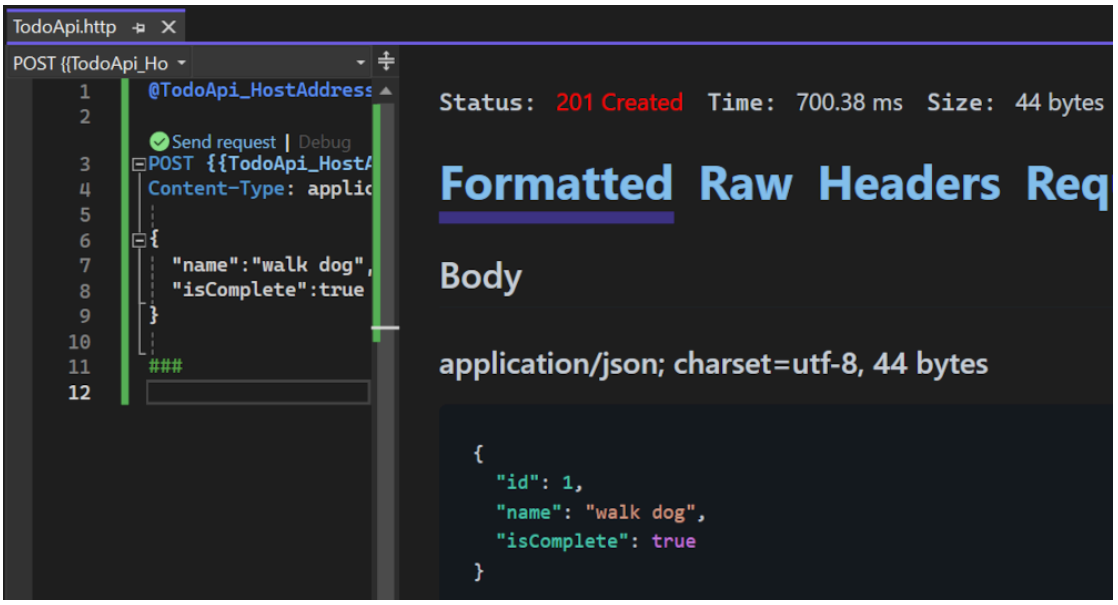
The preceding code adds a Content-Type header and a JSON request body. The TodoApi.http file should now look like the following example, but with your port number:

```
@TodoApi_HostAddress = https://localhost:7057

Post {{TodoApi_HostAddress}}/todoitems
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}

###
```

- Run the app.
- Select the **Send request** link that is above the `POST` request line.



The POST request is sent to the app and the response is displayed in the **Response** pane.



Examine the GET endpoints

The sample app implements several GET endpoints by calling `MapGet`:

Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get all completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item

C#

```
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

Test the GET endpoints

Visual Studio

Test the app by calling the `GET` endpoints from a browser or by using **Endpoints Explorer**. The following steps are for **Endpoints Explorer**.

- In **Endpoints Explorer**, right-click the first **GET** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Get {{TodoApi_HostAddress}}/todoitems

  ###
  ```

- Select the **Send request** link that is above the new `GET` request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  JSON

  ```
  [
    {
      "id": 1,
      "name": "walk dog",
      "isComplete": true
    }
  ]
  ```

- In **Endpoints Explorer**, right-click the `/todoitems/{id}` **GET** endpoint and select **Generate request**. The following content is added to the `TodoApi.http` file:

  ```
  GET {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- Replace `{id}` with `1`.

- Select the **Send request** link that is above the new GET request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  ```
  JSON

  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
  ```

This app uses an in-memory database. If the app is restarted, the GET request doesn't return any data. If no data is returned, POST data to the app and try the GET request again.

Return values

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

The return types can represent a wide range of HTTP status codes. For example, `GET /todoitems/{id}` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Examine the PUT endpoint

The sample app implements a single PUT endpoint using `MapPut`:

```
C#

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
```

```
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

This method is similar to the `MapPost` method, except it uses HTTP PUT. A successful response returns 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PUT endpoint

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has `Id = 1` and set its name to `"feed fish"`.

Visual Studio

- In **Endpoints Explorer**, right-click the **PUT** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Put {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- In the PUT request line, replace `{id}` with `1`.
- Add the following lines immediately after the PUT request line:

  ```
  ```

```
Content-Type: application/json

{
  "name": "feed fish",
  "isComplete": false
}
```

The preceding code adds a Content-Type header and a JSON request body.

- Select the **Send request** link that is above the new PUT request line.

  The PUT request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Examine and test the DELETE endpoint

The sample app implements a single DELETE endpoint using `MapDelete`:

C#

```
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

Visual Studio

- In **Endpoints Explorer**, right-click the **DELETE** endpoint and select **Generate request**.

  A DELETE request is added to `TodoApi.http`.

- Replace `{id}` in the DELETE request line with `1`. The DELETE request should look like the following example:

  ```
  
  ```

```
DELETE {{TodoApi_HostAddress}}/todoitems/1

###
```

- Select the **Send request** link for the DELETE request.

  The DELETE request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Use the MapGroup API

The sample app code repeats the `todoitems` URL prefix each time it sets up an endpoint. APIs often have groups of endpoints with a common URL prefix, and the MapGroup method is available to help organize such groups. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata.

Replace the contents of `Program.cs` with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", async (TodoDb db) =>
    await db.Todos.ToListAsync());

todoItems.MapGet("/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

todoItems.MapGet("/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

```
todoItems.MapPost("/", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

todoItems.MapPut("/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

todoItems.MapDelete("/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The preceding code has the following changes:

- Adds `var todoItems = app.MapGroup("/todoitems");` to set up the group using the URL prefix `/todoitems`.
- Changes all the `app.Map<HttpVerb>` methods to `todoItems.Map<HttpVerb>`.
- Removes the URL prefix `/todoitems` from the `Map<HttpVerb>` method calls.

Test the endpoints to verify that they work the same.

Use the TypedResults API

Returning TypedResults rather than Results has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the

endpoint. For more information, see TypedResults vs Results.

The `Map<HttpVerb>` methods can call route handler methods instead of using lambdas. To see an example, update *Program.cs* with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();
```

```
    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

The `Map<HttpVerb>` code now calls methods instead of lambdas:

C#

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

These methods return objects that implement IResult and are defined by TypedResults:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

```
static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Unit tests can call these methods and test that they return the correct type. For example, if the method is `GetAllTodos`:

```
C#
```

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

Unit test code can verify that an object of type Ok<Todo[]> is returned from the handler method. For example:

C#

```
public async Task GetAllTodos_ReturnsOkOfTodosResult()
{
    // Arrange
    var db = CreateDbContext();

    // Act
    var result = await TodosApi.GetAllTodos(db);

    // Assert: Check for the correct returned type
    Assert.IsType<Ok<Todo[]>>(result);
}
```

Prevent over-posting

Currently the sample app exposes the entire `Todo` object. Production apps In production applications, a subset of the model is often used to restrict the data that can be input and returned. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO can be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `Todo` class to include a secret field:

C#

```
public class Todo
{
    public int Id { get; set; }
```

```
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public string? Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a file named `TodoItemDTO.cs` with the following code:

C#

```
public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name,
todoItem.IsComplete);
}
```

Replace the contents of the `Program.cs` file with the following code to use this DTO model:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

RouteGroupBuilder todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

```
app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Select(x => new
TodoItemDTO(x)).ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db) {
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).Select(x => new TodoItemDTO(x)).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(new TodoItemDTO(todo))
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(TodoItemDTO todoItemDTO, TodoDb
db)
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    db.Todos.Add(todoItem);
    await db.SaveChangesAsync();

    todoItemDTO = new TodoItemDTO(todoItem);

    return TypedResults.Created($"/todoitems/{todoItem.Id}",
todoItemDTO);
}

static async Task<IResult> UpdateTodo(int id, TodoItemDTO todoItemDTO,
TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}
```

```
static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Verify you can post and get all fields except the secret field.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

- Configure JSON serialization options.
- Handle errors and exceptions: The developer exception page is enabled by default in the development environment for minimal API apps. For information about how to handle errors and exceptions, see Handle errors in ASP.NET Core APIs.
- For an example of testing a minimal API app, see this GitHub sample.
- OpenAPI support in minimal APIs.
- Quickstart: Publish to Azure.
- Organizing ASP.NET Core Minimal APIs.

Learn more

See Minimal APIs quick reference

Tutorial: Get started with ASP.NET Core SignalR

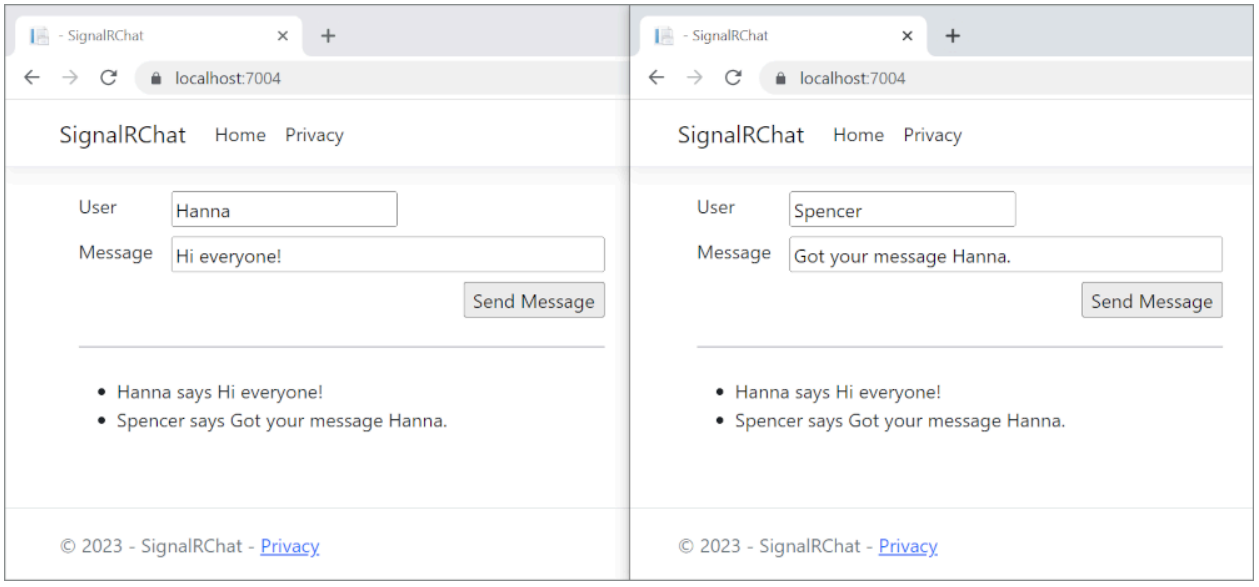
Article • 11/16/2023

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- ✓ Create a web project.
- ✓ Add the SignalR client library.
- ✓ Create a SignalR hub.
- ✓ Configure the project to use SignalR.
- ✓ Add code that sends messages from any client to all connected clients.

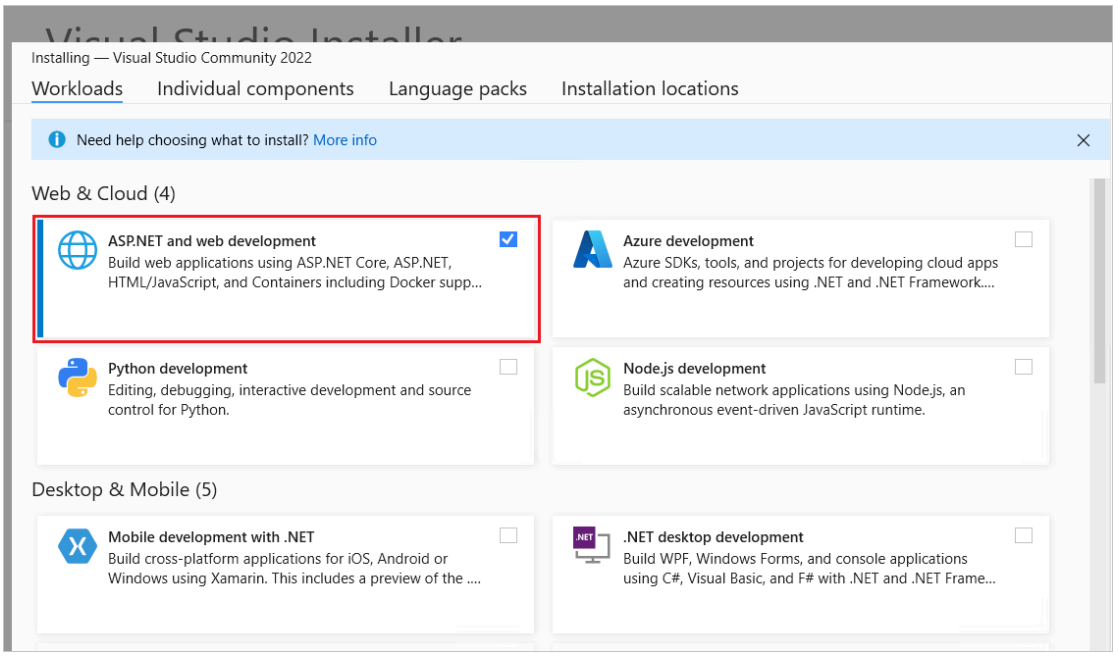
At the end, you'll have a working chat app:



Prerequisites

Visual Studio

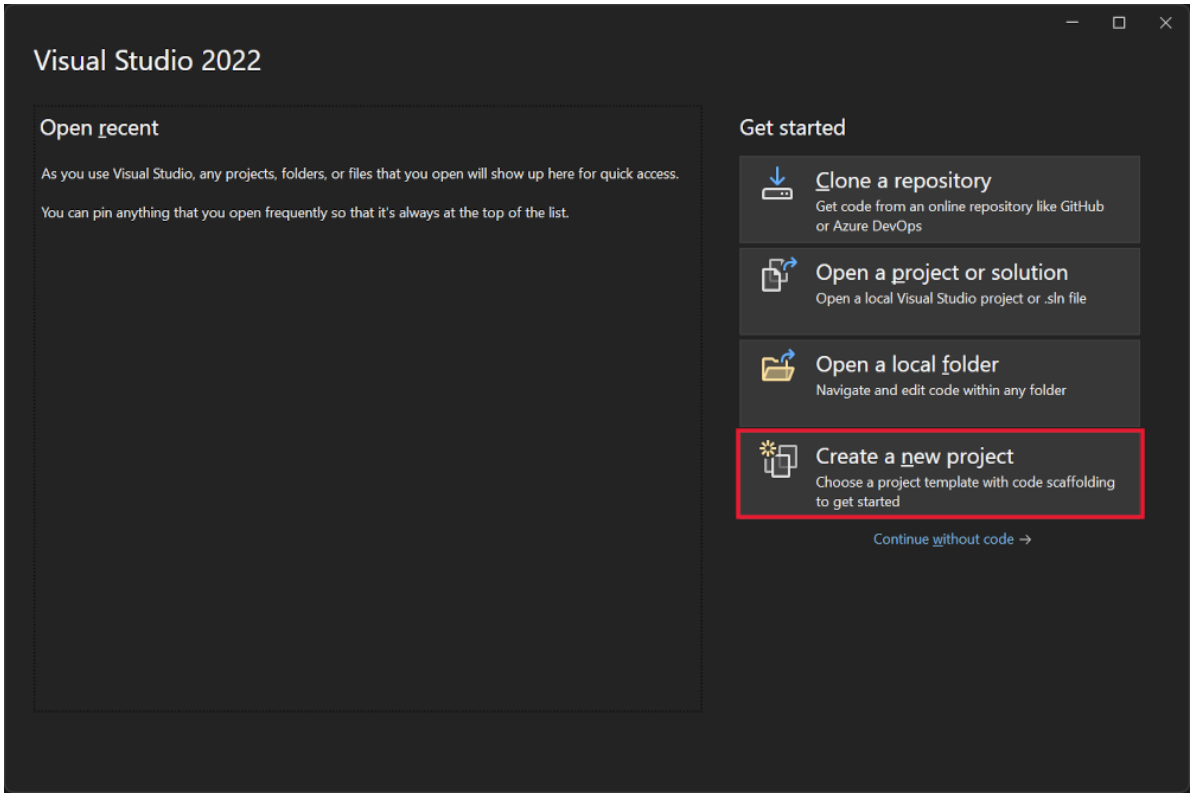
- Visual Studio 2022 with the **ASP.NET and web development** workload.



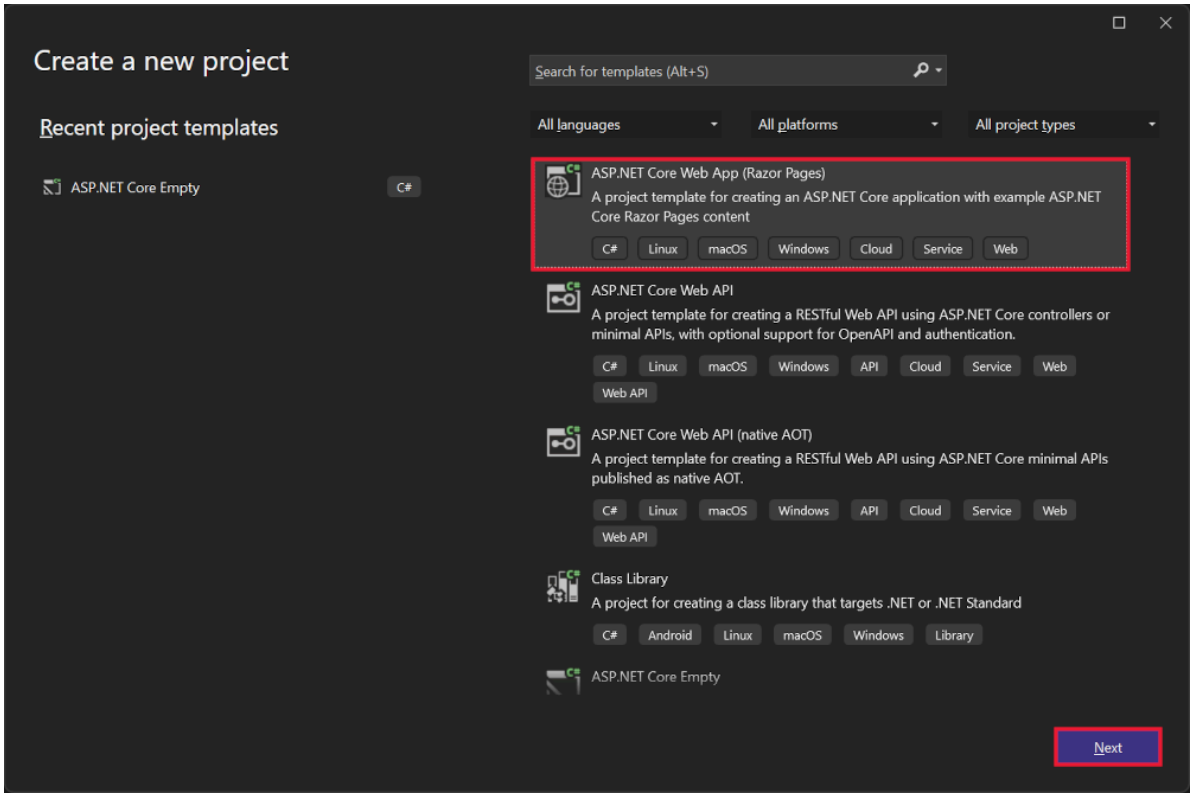
Create a web app project

Visual Studio

Start Visual Studio 2022 and select **Create a new project**.



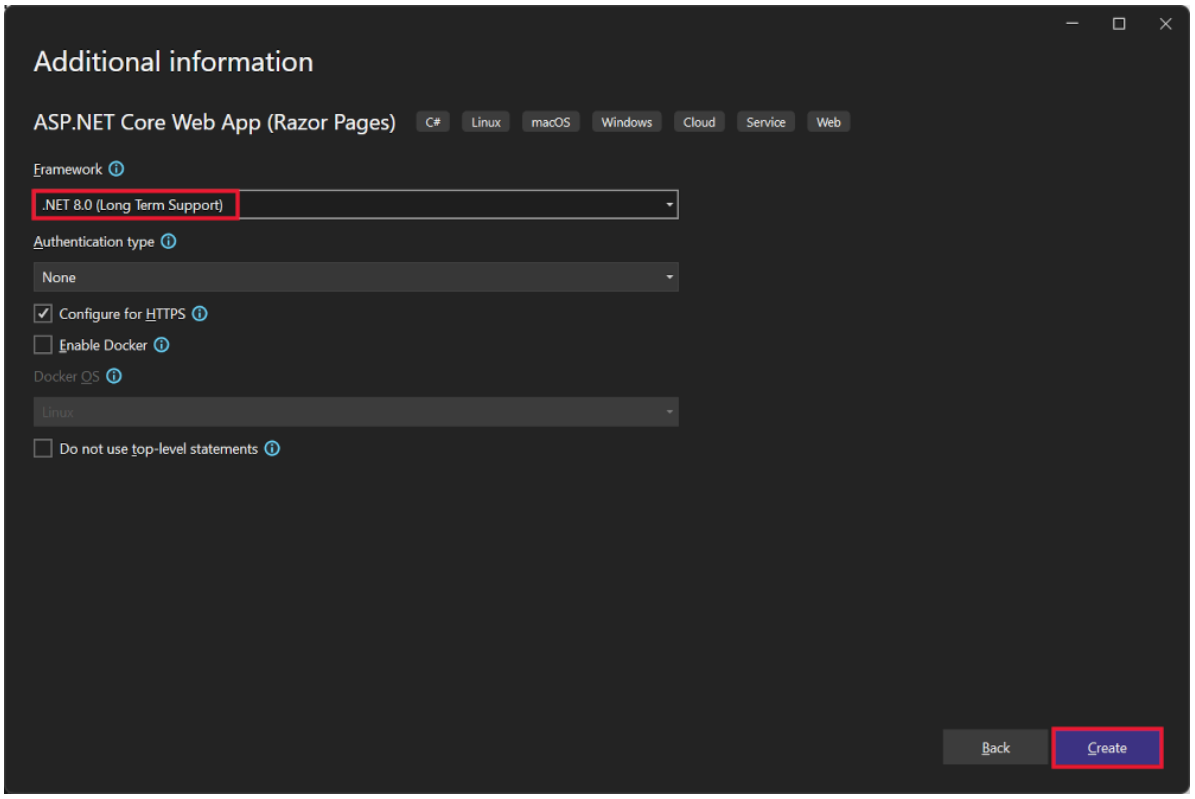
In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)**, and then select **Next**.



In the **Configure your new project** dialog, enter `SignalRChat` for **Project name**. It's important to name the project `SignalRChat`, including matching the capitalization, so the namespaces match the code in the tutorial.

Select **Next**.

In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.



Add the SignalR client library

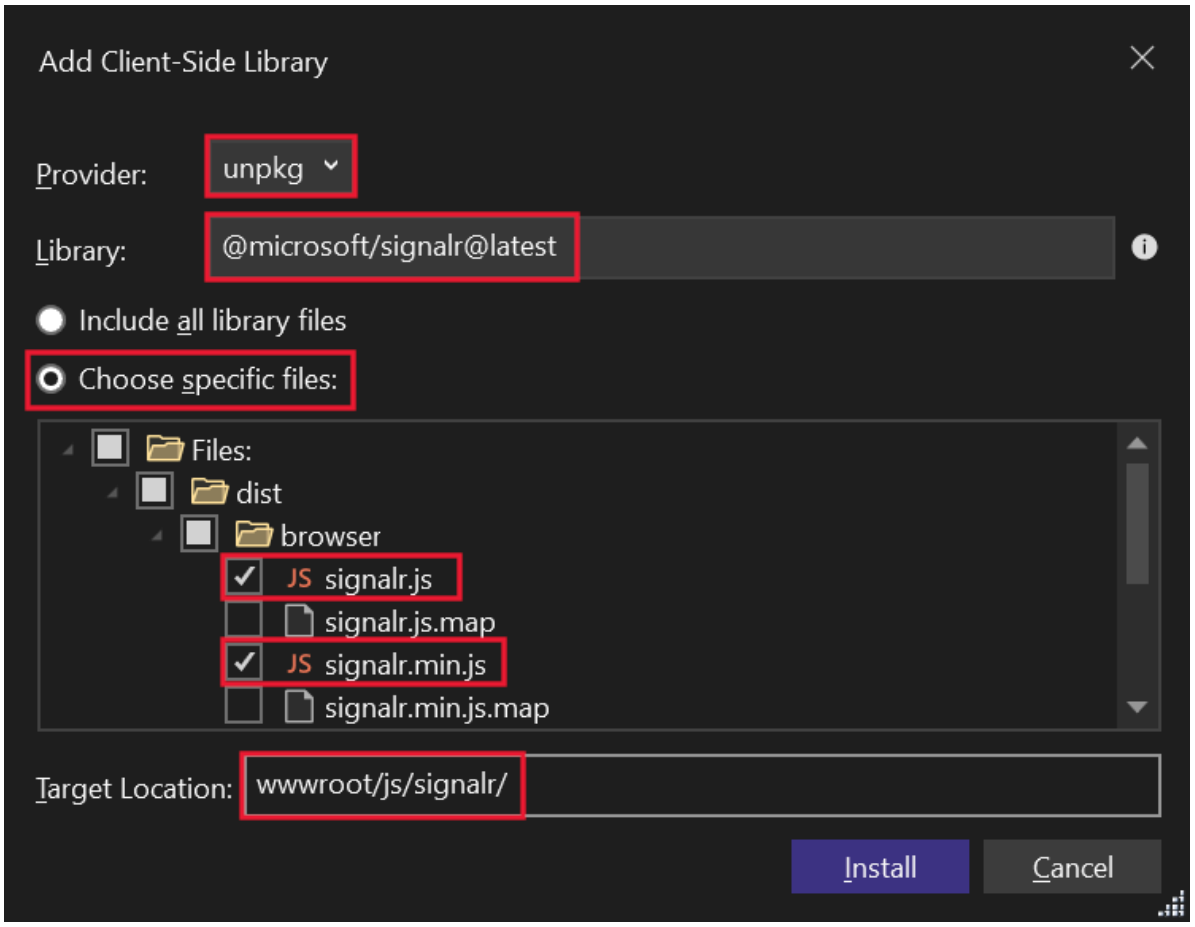
The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, use Library Manager (LibMan) to get the client library from unpkg. `unpkg` is a fast, global content delivery network for everything on npm.

Visual Studio

In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.

In the **Add Client-Side Library** dialog:

- Select **unpkg** for **Provider**
- Enter `@microsoft/signalr@latest` for **Library**.
- Select **Choose specific files**, expand the *dist/browser* folder, and select `signalr.js` and `signalr.min.js`.
- Set **Target Location** to `wwwroot/js/signalr/`.
- Select **Install**.



LibMan creates a `wwwroot/js/signalr` folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

In the SignalRChat project folder, create a `Hubs` folder.

In the `Hubs` folder, create the `ChatHub` class with the following code:

C#

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
```

```
    }
}
```

The `ChatHub` class inherits from the SignalR Hub class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the `Program.cs` file.

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

The preceding highlighted code adds SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

Replace the content in `Pages/Index.cshtml` with the following code:

CSHTML

```
@page
<div class="container">
    <div class="row p-1">
        <div class="col-1">User</div>
        <div class="col-5"><input type="text" id="userInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-1">Message</div>
        <div class="col-5"><input type="text" class="w-100"
id="messageInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-6 text-end">
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <hr />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

The preceding markup:

- Creates text boxes and a submit button.
- Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
- Includes script references to SignalR and the `chat.js` app code is created in the next step.

In the `wwwroot/js` folder, create a `chat.js` file with the following code:

JavaScript

```
"use strict";

var connection = new
signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable the send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    // We can assign user-supplied strings to an element's textContent
because it
    // is not interpreted as markup. If you're assigning in any other way,
you
    // should be aware of possible script injection concerns.
    li.textContent = `${user} says ${message}`;
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding JavaScript:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

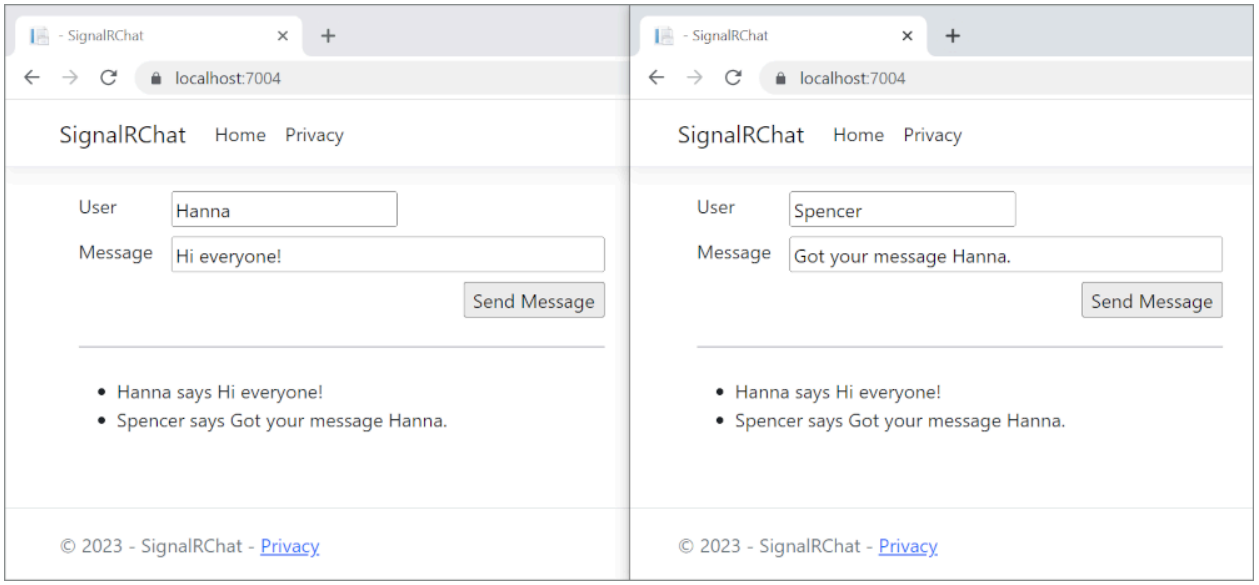
Visual Studio

Select `Ctrl`+`F5` to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

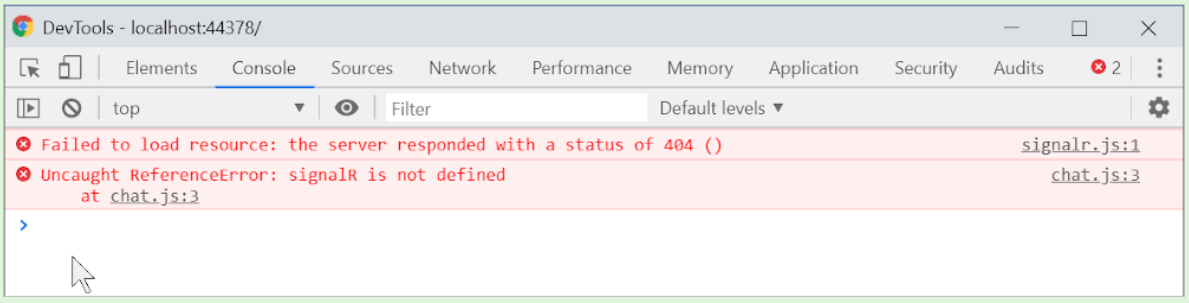
Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.



Tip

If the app doesn't work, open the browser developer tools (F12) and go to the console. Look for possible errors related to HTML and JavaScript code. For example, if `signalr.js` was put in a different folder than directed, the reference to that file won't work resulting in a 404 error in the console.



If an `ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY` error has occurred in Chrome, run the following commands to update the development certificate:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app. For more information on Azure SignalR Service, see What is Azure SignalR Service?.

Next steps

- Use hubs
- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- View or download sample code (how to download)

Tutorial: Get started with ASP.NET Core SignalR using TypeScript and Webpack

Article • 11/16/2023

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Sébastien Sougnez

This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app to bundle and build a client written in TypeScript. Webpack enables developers to bundle and build the client-side resources of a web app.

In this tutorial, you learn how to:

- ✓ Create an ASP.NET Core SignalR app
- ✓ Configure the SignalR server
- ✓ Configure a build pipeline using Webpack
- ✓ Configure the SignalR TypeScript client
- ✓ Enable communication between the client and the server

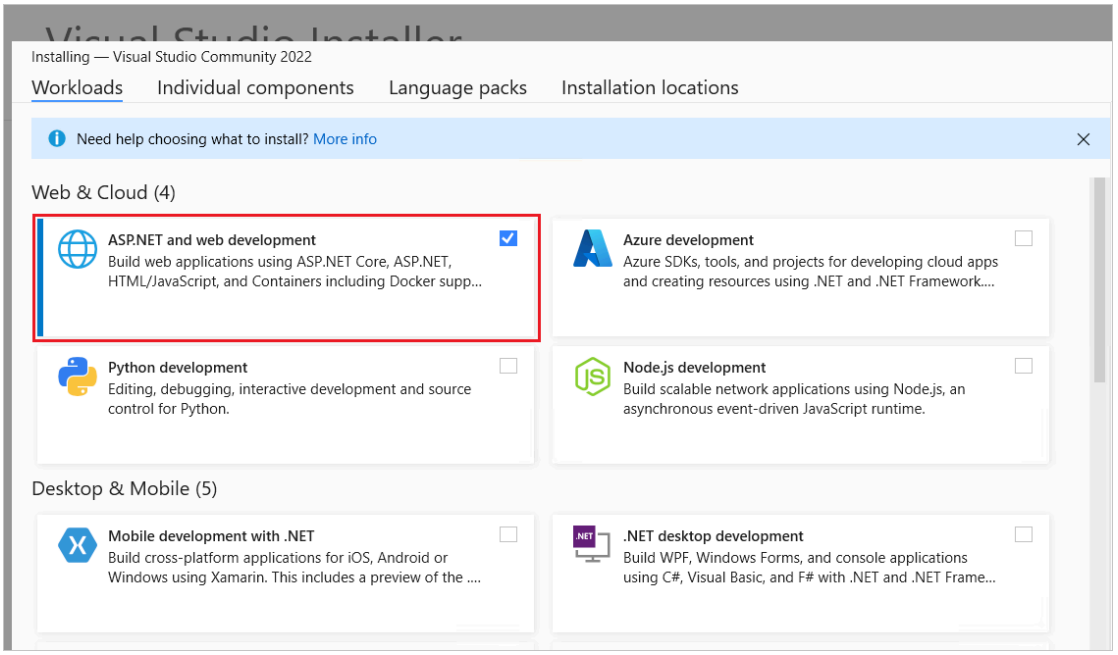
View or download sample code (how to download)

Prerequisites

- Node.js with npm

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



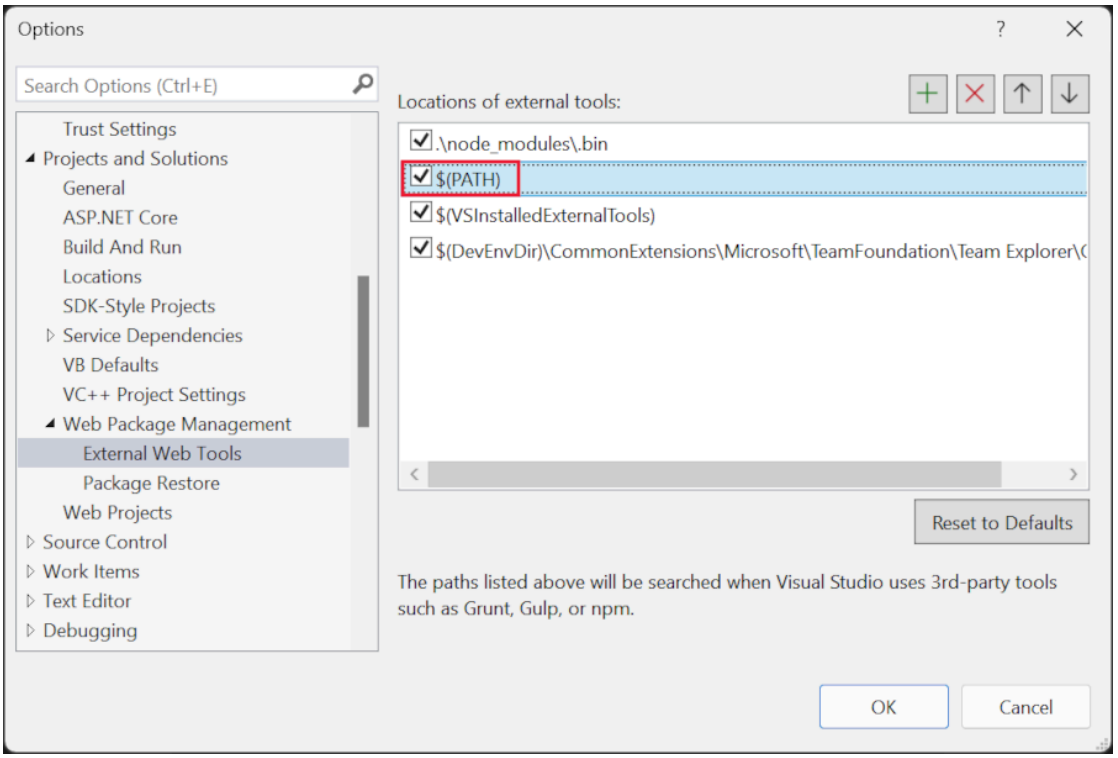
Create the ASP.NET Core web app

Visual Studio

By default, Visual Studio uses the version of npm found in its installation directory. To configure Visual Studio to look for npm in the `PATH` environment variable:

Launch Visual Studio. At the start window, select **Continue without code**.

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
2. Select the `$(PATH)` entry from the list. Select the up arrow to move the entry to the second position in the list, and select **OK**:



.

To create a new ASP.NET Core web app:

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Empty** template. Select **Next**.
2. Name the project `SignalRWebpack`, and select **Create**.
3. Select **.NET 8.0 (Long Term Support)** from the **Framework** drop-down. Select **Create**.

Add the Microsoft.TypeScript.MSBuild NuGet package to the project:

1. In **Solution Explorer**, right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild` and then select **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure the server

In this section, you configure the ASP.NET Core web app to send and receive SignalR messages.

1. In `Program.cs`, call AddSignalR:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
```

2. Again, in `Program.cs`, call UseDefaultFiles and UseStaticFiles:

```
C#

var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();
```

   The preceding code allows the server to locate and serve the `index.html` file. The file is served whether the user enters its full URL or the root URL of the web app.

3. Create a new directory named `Hubs` in the project root, `SignalRWebpack/`, for the SignalR hub class.

4. Create a new file, `Hubs/ChatHub.cs`, with the following code:

```
C#

using Microsoft.AspNetCore.SignalR;

namespace SignalRWebpack.Hubs;

public class ChatHub : Hub
{
    public async Task NewMessage(long username, string message) =>
        await Clients.All.SendAsync("messageReceived", username,
message);
}
```

   The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name is enough.

   In this example:

   - The TypeScript client sends a message identified as `newMessage`.
   - The C# `NewMessage` method expects the data sent by the client.
   - A call is made to SendAsync on Clients.All.
   - The received messages are sent to all clients connected to the hub.

5. Add the following `using` statement at the top of `Program.cs` to resolve the `ChatHub` reference:

```C#
using SignalRWebpack.Hubs;
```

6. In `Program.cs`, map the `/hub` route to the `ChatHub` hub. Replace the code that displays `Hello World!` with the following code:

```C#
app.MapHub<ChatHub>("/hub");
```

Configure the client

In this section, you create a Node.js project to convert TypeScript to JavaScript and bundle client-side resources, including HTML and CSS, using Webpack.

1. Run the following command in the project root to create a `package.json` file:

```Console
npm init -y
```

2. Add the highlighted property to the `package.json` file and save the file changes:

```JSON
{
  "name": "SignalRWebpack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
Console

npm i -D -E clean-webpack-plugin css-loader html-webpack-plugin mini-
css-extract-plugin ts-loader typescript webpack webpack-cli
```

The `-E` option disables npm's default behavior of writing semantic versioning range operators to `package.json`. For example, `"webpack": "5.76.1"` is used instead of `"webpack": "^5.76.1"`. This option prevents unintended upgrades to newer package versions.

For more information, see the npm-install documentation.

4. Replace the `scripts` property of `package.json` file with the following code:

```
JSON

"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

The following scripts are defined:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. use `build` in development only.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET CLI's publish command to publish the app.

5. Create a file named `webpack.config.js` in the project root, with the following code:

```
JavaScript

const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
```

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/",
    },
    resolve: {
        extensions: [".js", ".ts"],
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader",
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"],
            },
        ],
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html",
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css",
        }),
    ],
};
```

The preceding file configures the Webpack compilation process:

- The `output` property overrides the default value of `dist`. The bundle is instead emitted in the `wwwroot` directory.
- The `resolve.extensions` array includes `.js` to import the SignalR client JavaScript.

6. Create a new directory named `src` in the project root, `SignalRWebpack/`, for the client code.

7. Copy the `src` directory and its contents from the sample project into the project root. The `src` directory contains the following files:

    - `index.html`, which defines the homepage's boilerplate markup:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR with TypeScript and
Webpack</title>
  </head>
  <body>
    <div id="divMessages" class="messages"></div>
    <div class="input-zone">
      <label id="lblMessage" for="tbMessage">Message:</label>
      <input id="tbMessage" class="input-zone-input" type="text"
/>
      <button id="btnSend">Send</button>
    </div>
  </body>
</html>
```

- `css/main.css`, which provides CSS styles for the homepage:

css

```
*,
*::before,
*::after {
  box-sizing: border-box;
}

html,
body {
  margin: 0;
  padding: 0;
}

.input-zone {
  align-items: center;
  display: flex;
  flex-direction: row;
  margin: 10px;
}

.input-zone-input {
  flex: 1;
  margin-right: 10px;
}

.message-author {
  font-weight: bold;
}

.messages {
```

```
  border: 1px solid #000;
  margin: 10px;
  max-height: 300px;
  min-height: 300px;
  overflow-y: auto;
  padding: 5px;
}
```

- `tsconfig.json`, which configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript:

JSON

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

- `index.ts`:

TypeScript

```
import * as signalR from "@microsoft/signalr";
import "./css/main.css";

const divMessages: HTMLDivElement =
document.querySelector("#divMessages");
const tbMessage: HTMLInputElement =
document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement =
document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message:
string) => {
  const m = document.createElement("div");

  m.innerHTML = `<div class="message-author">${username}</div>
<div>${message}</div>`;

  divMessages.appendChild(m);
  divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch((err) => document.write(err));
```

```
tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
  if (e.key === "Enter") {
    send();
  }
});

btnSend.addEventListener("click", send);

function send() {
  connection.send("newMessage", username, tbMessage.value)
    .then(() => (tbMessage.value = ""));
}
```

The preceding code retrieves references to DOM elements and attaches two event handlers:

- `keyup`: Fires when the user types in the `tbMessage` textbox and calls the `send` function when the user presses the **Enter** key.
- `click`: Fires when the user selects the **Send** button and calls `send` function is called.

The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. It's added to the main `div` element displaying the messages.

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

8. Run the following command at the project root:

```
Console
```

```
npm i @microsoft/signalr @types/node
```

The preceding command installs:

- The SignalR TypeScript client, which allows the client to send messages to the server.
- The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

Test the app

Confirm that the app works with the following steps:

Visual Studio

1. Run Webpack in `release` mode. Using the **Package Manager Console** window, run the following command in the project root.

   Console

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the `wwwroot` folder.

   Webpack completed the following tasks:

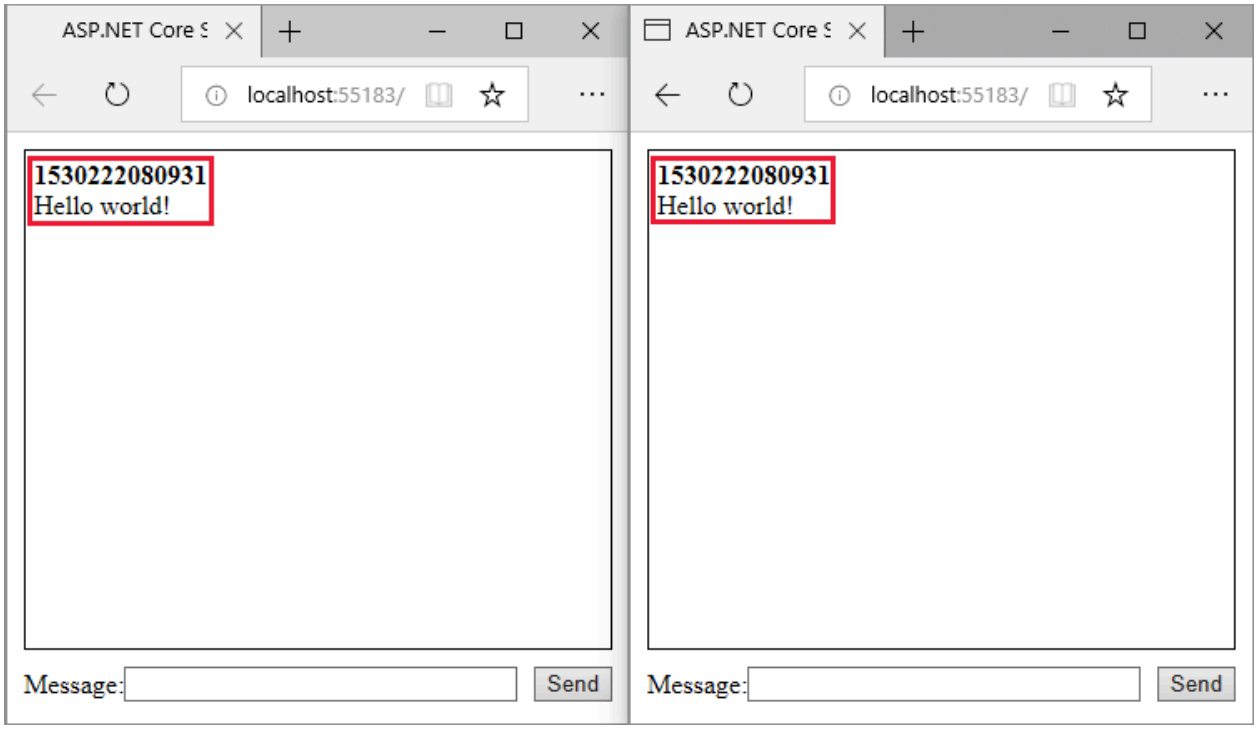
   - Purged the contents of the `wwwroot` directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from `src` to the `wwwroot` directory.
   - Injected the following elements into the `wwwroot/index.html` file:
     - A `<link>` tag, referencing the `wwwroot/main.<hash>.css` file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified `wwwroot/main.<hash>.js` file. This tag is placed immediately after the closing `</title>` tag.

2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The `wwwroot/index.html` file is served at `https://localhost:<port>`.

   If there are compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser) and paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and select the **Send** button. The unique user name and message are displayed on both pages instantly.



Next steps

- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- MessagePack Hub Protocol in SignalR for ASP.NET Core

Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Use ASP.NET Core SignalR with Blazor

Article • 06/21/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected

- Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
```

```
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
```

```
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> ⓘ Note
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

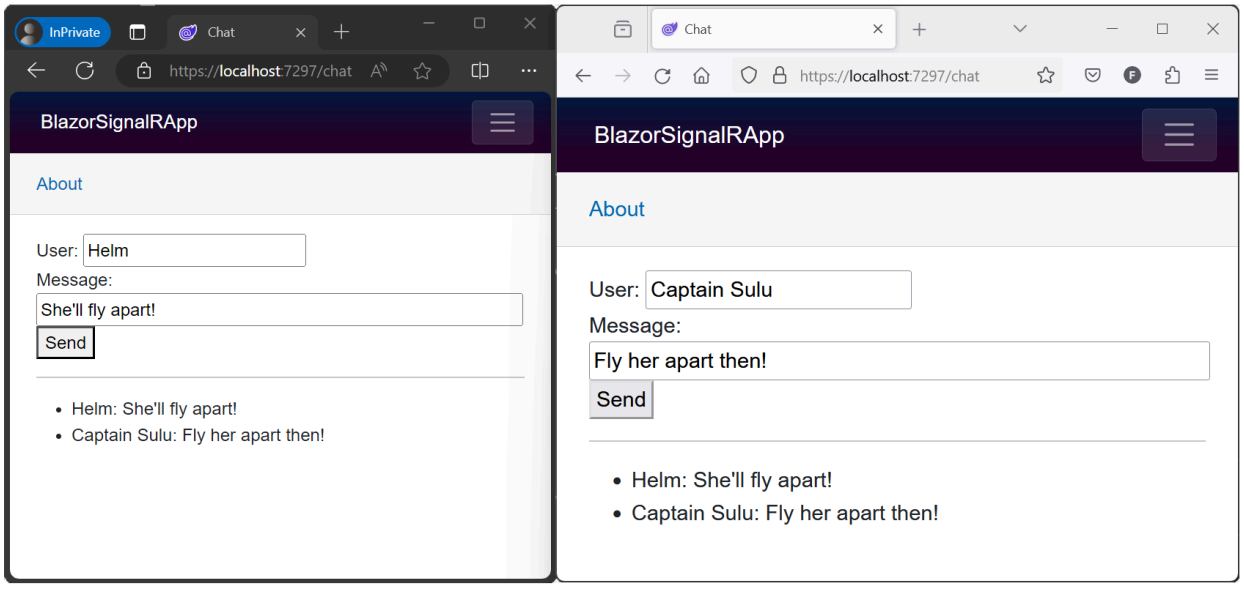
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Tutorial: Create a gRPC client and server in ASP.NET Core

Article • 06/18/2024

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server. At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

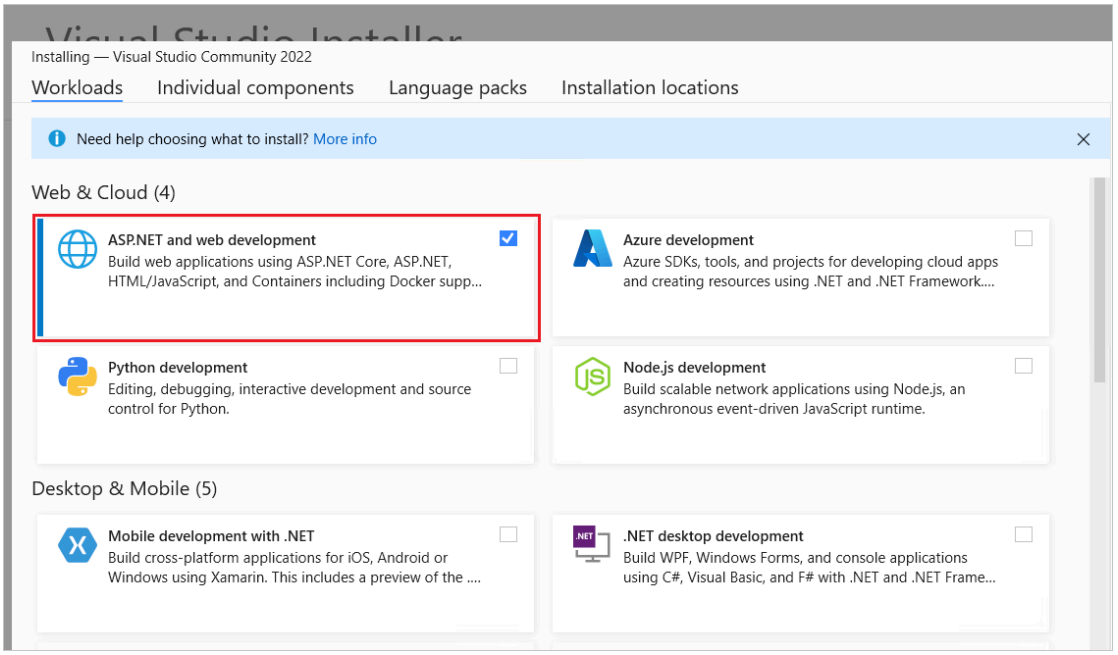
In this tutorial, you:

- ✓ Create a gRPC Server.
- ✓ Create a gRPC client.
- ✓ Test the gRPC client with the gRPC Greeter service.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a gRPC service

Visual Studio

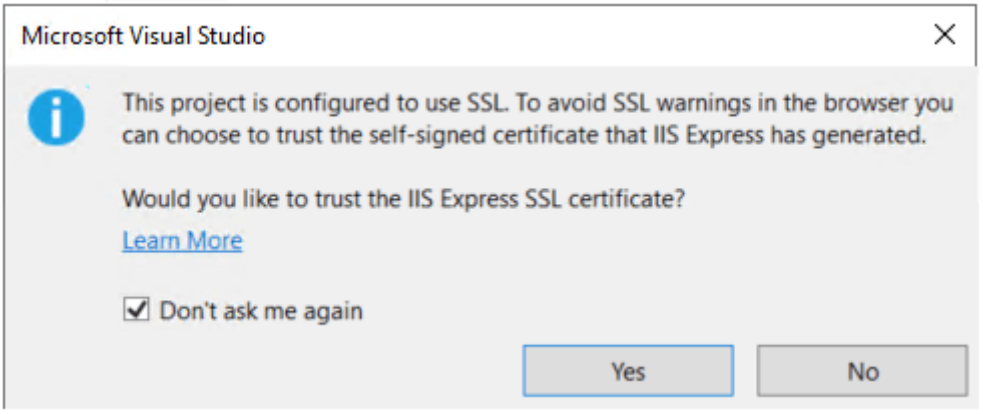
- Start Visual Studio 2022 and select **New Project**.
- In the **Create a new project** dialog, search for `gRPC`. Select **ASP.NET Core gRPC Service** and select **Next**.
- In the **Configure your new project** dialog, enter `GrpcGreeter` for **Project name**. It's important to name the project *GrpcGreeter* so the namespaces match when you copy and paste code.
- Select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Run the service

Visual Studio

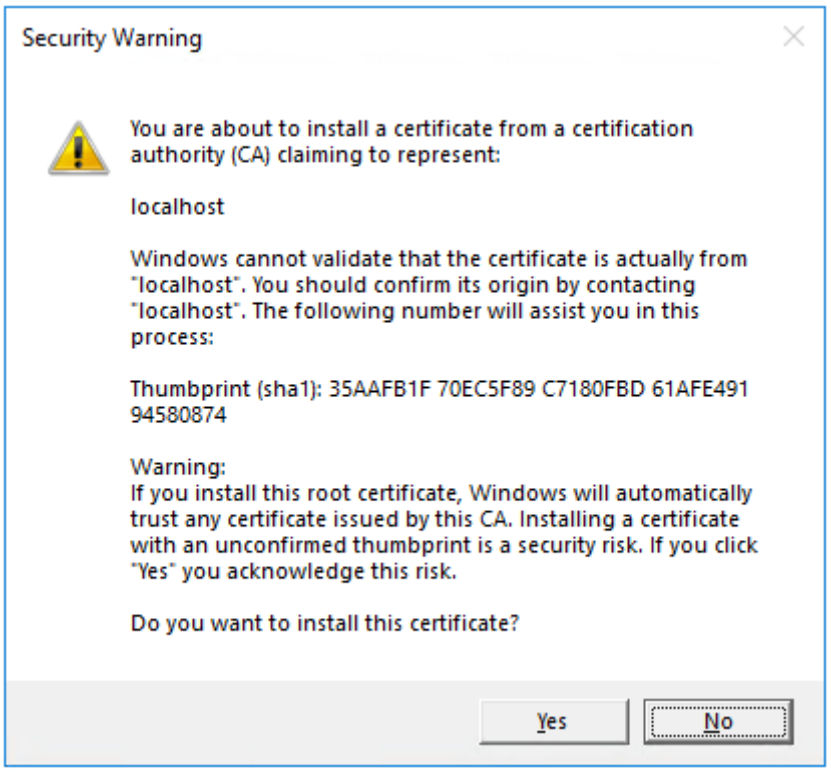
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:

  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Starts Kestrel server.
- Launches a browser.
- Navigates to `http://localhost:port`, such as `http://localhost:7042`.
  - *port*: A randomly assigned port number for the app.
  - `localhost`: The standard hostname for the local computer. Localhost only serves web requests from the local computer.

The logs show the service listening on `https://localhost:<port>`, where `<port>` is the localhost port number randomly assigned when the project is created and set in `Properties/launchSettings.json`.

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **Note**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server. The gRPC service localhost port number is randomly assigned when the project is created and set in the *Properties\launchSettings.json* file of the gRPC service project.

Examine the project files

GrpcGreeter project files:

- `Protos/greet.proto`: defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- `Services` folder: Contains the implementation of the `Greeter` service.
- `appSettings.json`: Contains configuration data such as the protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- `Program.cs`, which contains:
  - The entry point for the gRPC service. For more information, see .NET Generic Host in ASP.NET Core.
  - Code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

Visual Studio

- Open a second instance of Visual Studio and select **New Project**.
- In the **Create a new project** dialog, select **Console App**, and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Add required NuGet packages

The gRPC client project requires the following NuGet packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.

- Grpc.Tools, which contain C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

Visual Studio

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the `GrpcGreeterClient.csproj` files.
- Run the following commands:

PowerShell

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**.
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the *Protos* folder in the gRPC client project.
- Update the namespace inside the `greet.proto` file to the project's namespace:

JSON

```
option csharp_namespace = "GrpcGreeterClient";
```

- Edit the `GrpcGreeterClient.csproj` project file:

Visual Studio

Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

- Build the client project to create the types in the `GrpcGreeterClient` namespace.

> **Note**
>
> The `GrpcGreeterClient` types are generated automatically by the build process. The tooling package Grpc.Tools generates the following files based on the *greet.proto* file:
>
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\Greet.cs`: The protocol buffer code which populates, serializes and retrieves the request and response message types.
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\GreetGrpc.cs`: Contains the generated client classes.
>
> For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

- Update the gRPC client `Program.cs` file with the following code.

C#

```
using System.Threading.Tasks;
using Grpc.Net.Client;
using GrpcGreeterClient;

// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

- In the preceding highlighted code, replace the localhost port number `7042` with the `HTTPS` port number specified in `Properties/launchSettings.json` within the `GrpcGreeter` service project.

`Program.cs` contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
```

```
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

Test the gRPC client with the gRPC Greeter service

Update the `appsettings.Development.json` file by adding the following highlighted lines:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
      ,"Microsoft.AspNetCore.Hosting": "Information",
      "Microsoft.AspNetCore.Routing.EndpointMiddleware": "Information"
    }
  }
}
```

Visual Studio

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

Console

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path:
C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-
start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:
<port>/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **Note**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

Next steps

- View or download the completed sample code for this tutorial (how to download).
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

Article • 09/27/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Tom Dykstra, Jeremy Likness, and Jon P Smith

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Database engines

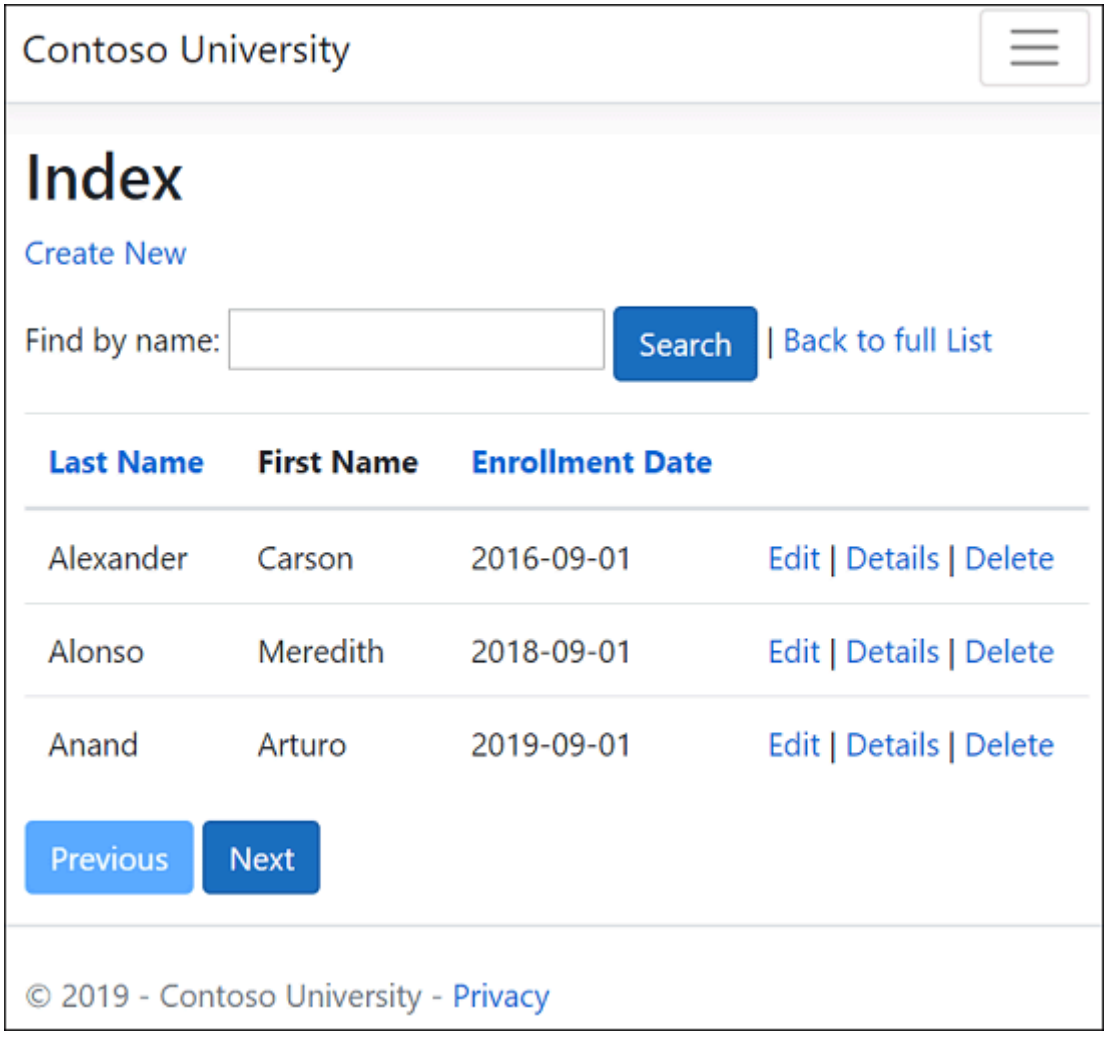
The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

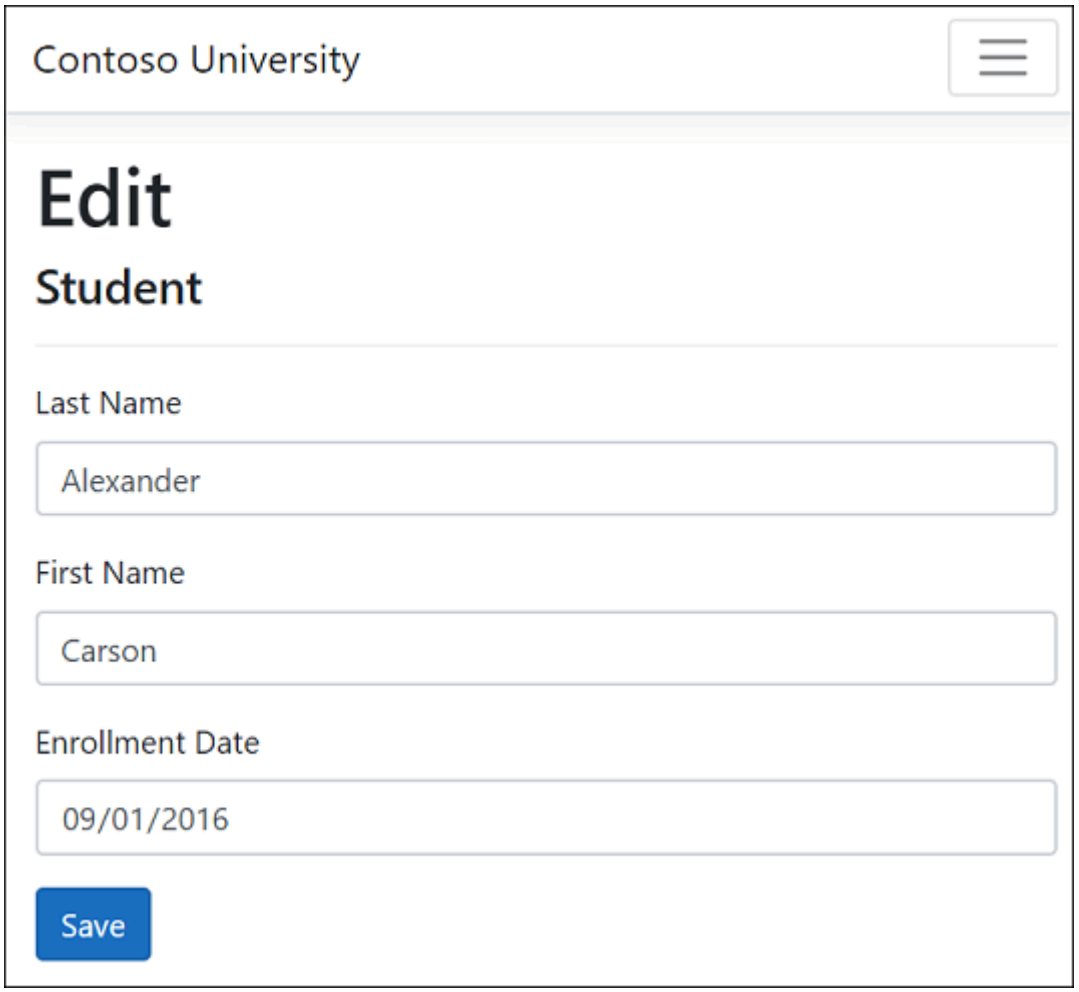
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core with ASP.NET Core, not how to customize the UI.

Optional: Build the sample download

This step is optional. Building the completed app is recommended when you have problems you can't solve. If you run into a problem you can't resolve, compare your code to the completed project. Download instructions.

Visual Studio

Select `ContosoUniversity.csproj` to open the project.

- Build the project.
- In Package Manager Console (PMC) run the following command:

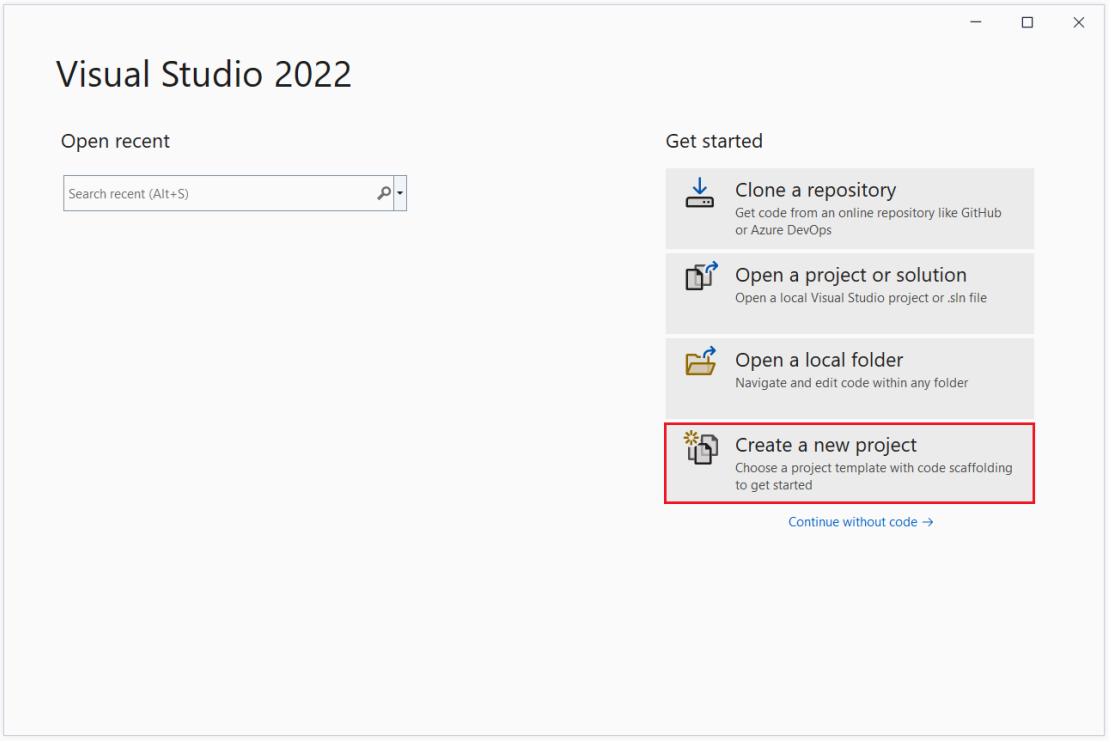
```
PowerShell
Update-Database
```

Run the project to seed the database.

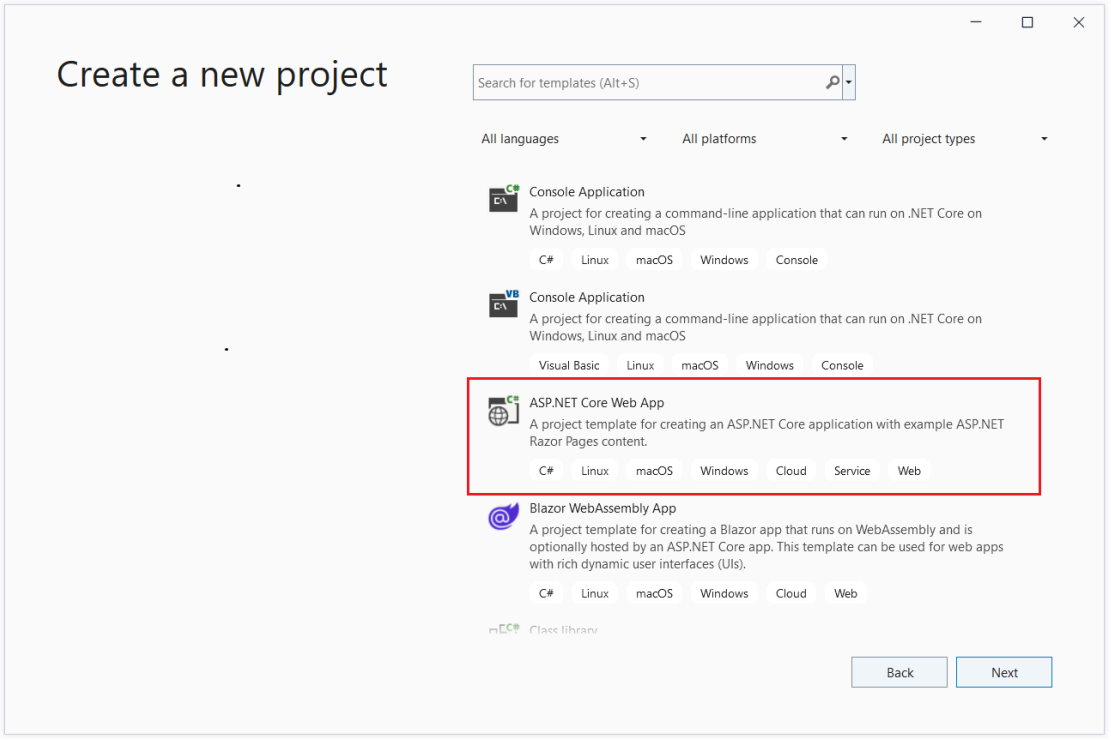
Create the web app project

Visual Studio

1. Start Visual Studio 2022 and select **Create a new project**.



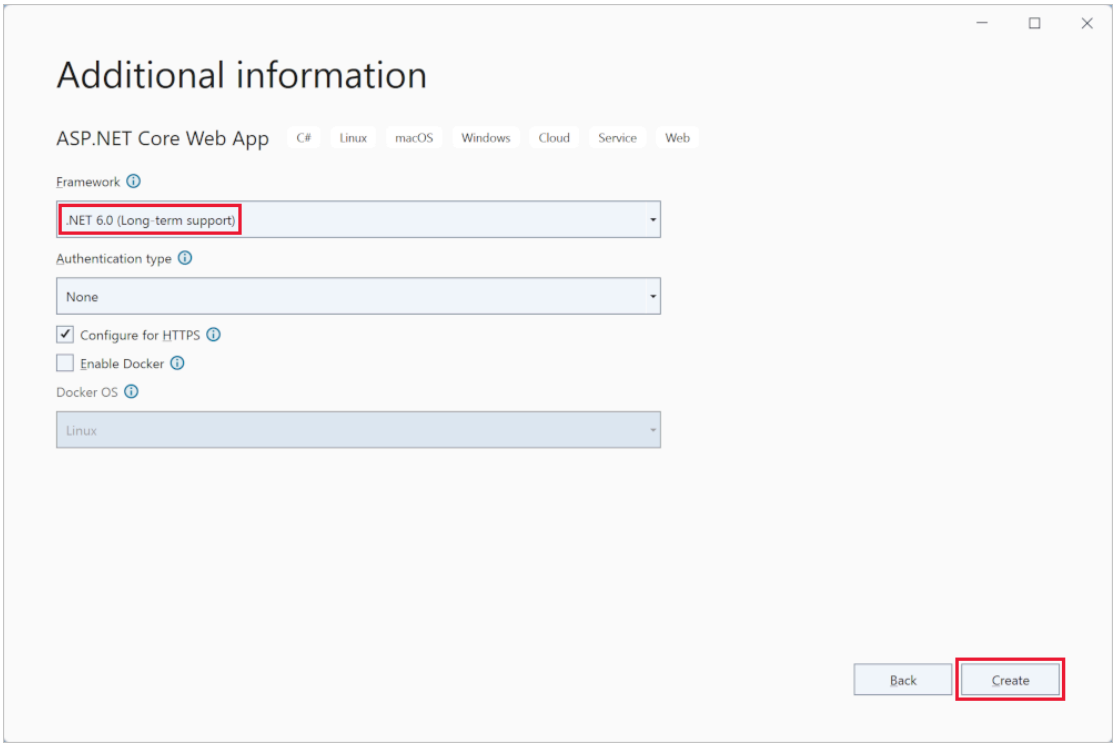
2. In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.



3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to name the project *ContosoUniversity*, including matching the capitalization, so the namespaces will match when you copy and paste example code.

4. Select **Next**.

5. In the **Additional information** dialog, select **.NET 6.0 (Long-term support)** and then select **Create**.



Set up the site style

Copy and paste the following code into the `Pages/Shared/_Layout.cshtml` file:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/ContosoUniversity.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
```

```
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Contoso University - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The layout file sets the site header, footer, and menu. The preceding code makes the following changes:

- Each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- The **Home** and **Privacy** menu entries are deleted.

- Entries are added for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

In `Pages/Index.cshtml`, replace the contents of the file with the following code:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in
a series of tutorials.
                </p>
                <p>
@*                      <a
href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro"
class="stretched-link">See the tutorial</a>
*@                  </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
@*                      <a
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-rp/intro/samples" class="stretched-link">See project source code</a>
*@                  </p>
            </div>
        </div>
```

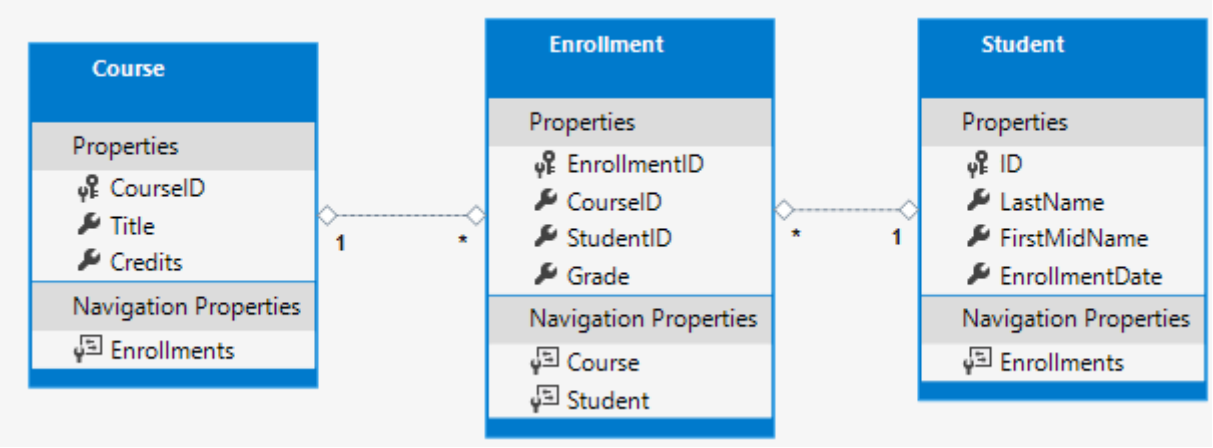
```
    </div>
</div>
```

The preceding code replaces the text about ASP.NET Core with text about this app.

Run the app to verify that the home page appears.

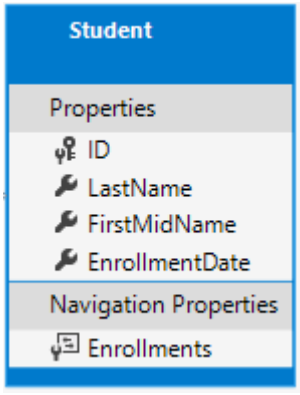
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
C#

namespace ContosoUniversity.Models
{
    public class Student
```

```
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

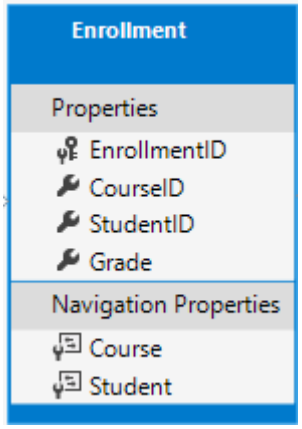
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its `StudentID` column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have `StudentID` = 1. `StudentID` is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. Other collection types can be used, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create `Models/Enrollment.cs` with the following code:

```
C#

using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, many developers choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

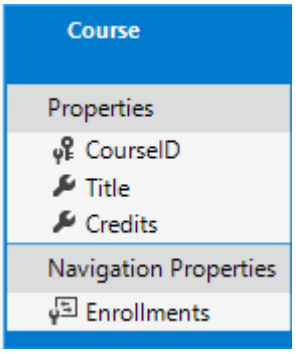
The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties

can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create `Models/Course.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the app. The compiler generates several warnings about how `null` values are handled. See this GitHub issue, Nullable reference types, and Tutorial: Express your design intent more clearly with nullable and non-nullable reference types for more information.

To eliminate the warnings from nullable reference types, remove the following line from the `ContosoUniversity.csproj` file:

XML

```
<Nullable>enable</Nullable>
```

The scaffolding engine currently does not support nullable reference types, therefore the models used in scaffold can't either.

Remove the `?` nullable reference type annotation from `public string? RequestId { get; set; }` in `Pages/Error.cshtml.cs` so the project builds without compiler warnings.

Scaffold Student pages

In this section, the ASP.NET Core scaffolding tool is used to generate:

- An EF Core `DbContext` class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the Microsoft.EntityFrameworkCore.DbContext class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

Visual Studio

- Create a *Pages/Students* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add New Scaffold Item** dialog:
  - In the left tab, select **Installed** > **Common** > **Razor Pages**
  - Select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the + (plus) sign.
    - Change the data context name to end in `SchoolContext` rather than `ContosoUniversityContext`. The updated context name: `ContosoUniversity.Data.SchoolContext`
    - Select **Add** to finish adding the data context class.
    - Select **Add** to finish the **Add Razor Pages** dialog.

The following packages are automatically installed:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

If the preceding step fails, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Program.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

The scaffolding tool generates a connection string in the `appsettings.json` file.

Visual Studio

The connection string specifies SQL Server LocalDB:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext-
0e9;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf*

files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The preceding code changes from the singular `DbSet<Student> Student` to the plural `DbSet<Student> Students`. To make the Razor Pages code match the new `DBSet` name, make a global change from: `_context.Student.` to: `_context.Students.`

There are 8 occurrences.

Because an entity set contains multiple entities, many developers prefer the `DBSet` property names should be plural.

The highlighted code:

- Creates a DbSet<TEntity> property for each entity set. In EF Core terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- Calls OnModelCreating. `OnModelCreating`:
  - Is called when `SchoolContext` has been initialized but before the model has been secured and used to initialize the context.
  - Is required because later in the tutorial the `Student` entity will have references to the other entities.

We hope to fix this issue in a future release.

Program.cs

ASP.NET Core is built with dependency injection. Services such as the `SchoolContext` are registered with dependency injection during app startup. Components that require these services, such as Razor Pages, are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

Visual Studio

The following highlighted lines were added by the scaffolder:

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration

system reads the connection string from the `appsettings.json` or the `appsettings.Development.json` file.

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter and UseMigrationsEndPoint as shown in the following code:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}
```

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

In the Package Manager Console, enter the following to add the NuGet package:

PowerShell

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for Entity Framework Core error pages. This middleware helps to detect and diagnose errors with Entity Framework Core migrations.

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment for EF migrations errors.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    // DbInitializer.Initialize(context);
}
```

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works early in development when the schema is rapidly evolving, as long as data doesn't need to be preserved. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, the database is deleted that was created by `EnsureCreated` and migrations is used. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
C#
```

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.P
arse("2019-09-01")},
                new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017
-09-01")},
                new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2016-09-01")},
                new
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2019-09-01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
```

```
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, remove `//` from the `DbInitializer.Initialize` line:

C#

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    DbInitializer.Initialize(context);
}
```

Visual Studio

- Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database -Confirm
```

- Respond with `Y` to delete the database.

- Restart the app.
- Select the Students page to see the seeded data.

View the database

Visual Studio

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB** > **Databases** > **SchoolContext-{GUID}**. The database name is generated from the context name provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous EF methods in ASP.NET Core web apps

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't doing work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
C#

public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

> ⚠ **Warning**
>
> The async implementation of Microsoft.Data.SqlClient has some known issues (#593, #601, and others). If you're seeing unexpected performance problems,

try using sync command execution instead, especially when dealing with large text or binary values.

Performance considerations

In general, a web page shouldn't be loading an arbitrary number of rows. A query should use paging or a limiting approach. For example, the preceding query could use `Take` to limit the rows returned:

C#

```
public async Task OnGetAsync()
{
    Student = await _context.Students.Take(10).ToListAsync();
}
```

Enumerating a large table in a view could return a partially constructed HTTP 200 response if a database exception occurs part way through the enumeration.

Paging is covered later in the tutorial.

For more information, see Performance considerations (EF).

Next steps

Use SQLite for development, SQL Server for production

Next tutorial

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the `Details` page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data must be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the `Student` data, without the `Enrollment` data:

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the

`Enrollment.Course` navigation property. These methods are examined in detail in the Read related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
```

```
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is

unnecessary if the query can only return one entity, as when it searches on a unique key.

- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

C#

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` is converted to `DateTime`.

Run the app, and create a student entity to test the Create page.

Overposting

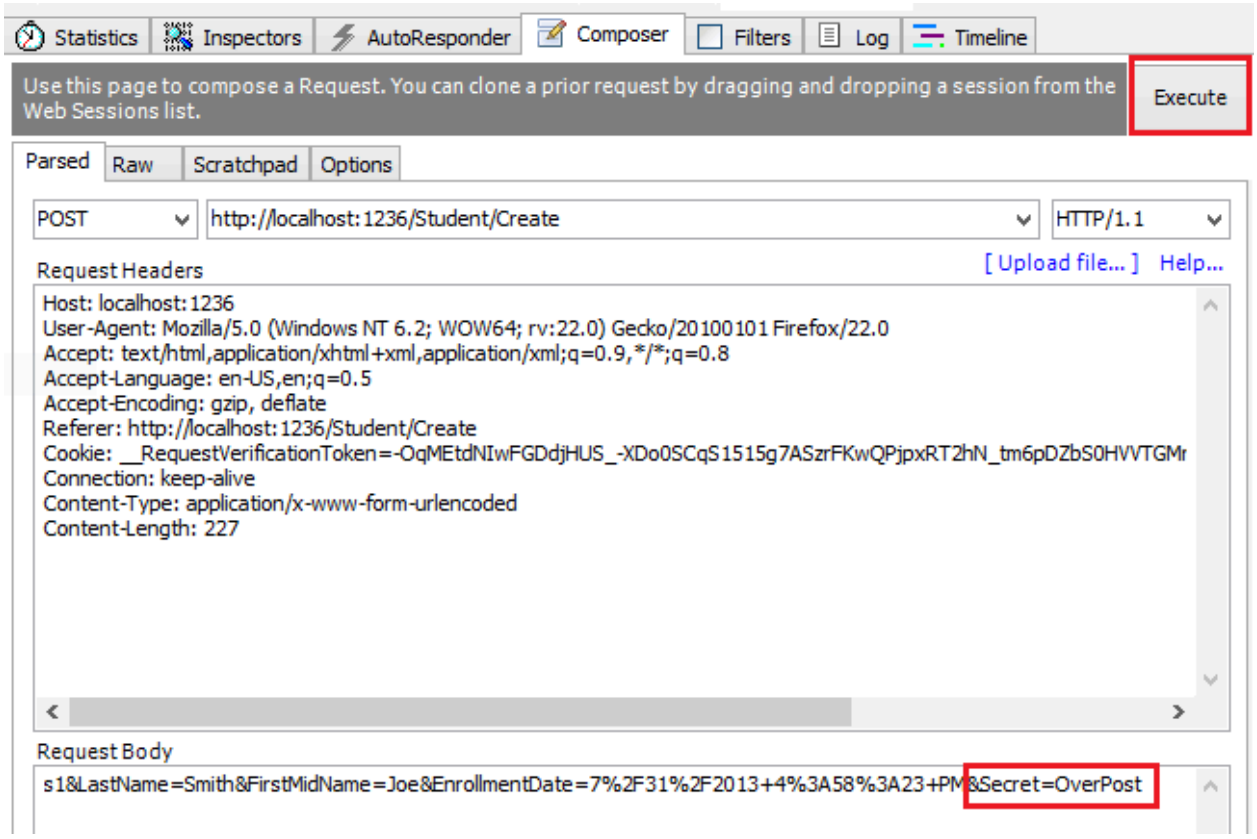
Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

C#

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field, with the value "OverPost", to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI page, for example, the Create page.

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `StudentVM` view model:

C#

```
public class StudentVM
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
```

```
    public DateTime EnrollmentDate { get; set; }
}
```

The following code uses the `StudentVM` view model to create a new student:

C#

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type:

- Doesn't need to be related to the model type.
- Needs to have properties that match.

Using `StudentVM` requires the Create page use `StudentVM` rather than `Student`:

CSHTML

```
@page
@model CreateVMModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
```

```
                <label asp-for="StudentVM.LastName" class="control-label">
</label>
                <input asp-for="StudentVM.LastName" class="form-control" />
                <span asp-validation-for="StudentVM.LastName" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.FirstMidName" class="control-
label"></label>
                <input asp-for="StudentVM.FirstMidName" class="form-control"
/>
                <span asp-validation-for="StudentVM.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.EnrollmentDate" class="control-
label"></label>
                <input asp-for="StudentVM.EnrollmentDate" class="form-
control" />
                <span asp-validation-for="StudentVM.EnrollmentDate"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

    if (studentToUpdate == null)
    {
        return NotFound();
    }

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an `INSERT` statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an `UPDATE` statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a `DELETE` statement.
- `Detached`: The entity isn't being tracked by the database context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL `UPDATE` statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, a custom error message is implemented when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
```

```
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<DeleteModel> _logger;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context,
                           ILogger<DeleteModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool?
saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Student == null)
            {
                return NotFound();
            }

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = String.Format("Delete {ID} failed. Try
again", id);
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }
```

```
            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException ex)
            {
                _logger.LogError(ex, ErrorMessage);

                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code:

- Adds Logging.
- Adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object.

The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is `false` when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` because the delete operation failed, the `saveChangesError` parameter is `true`.

The `OnPostAsync` method retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to `Pages/Students/Delete.cshtml`:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}
```

```
<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

Previous tutorial | Next tutorial

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

Article • 04/10/2024

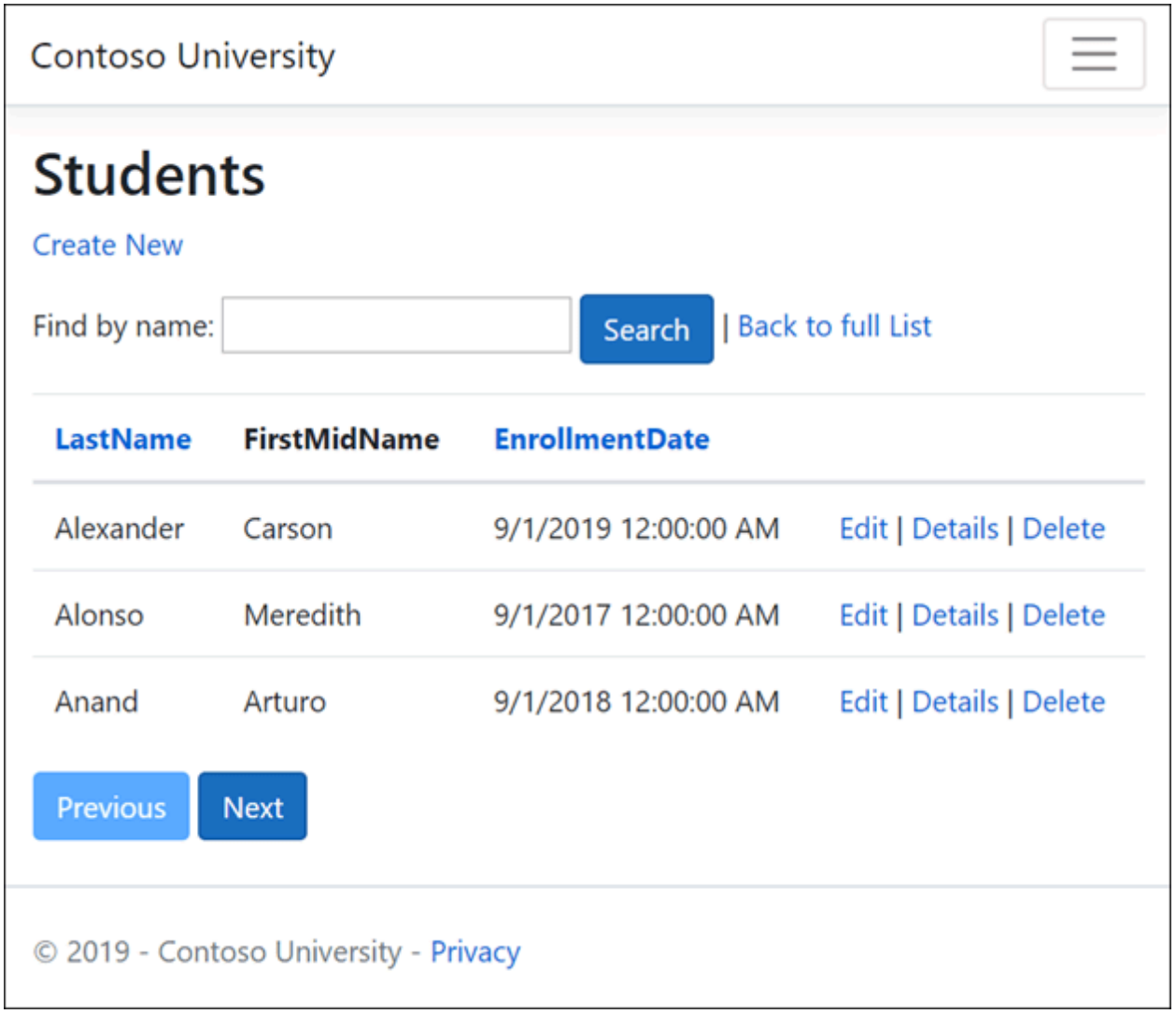
By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in `Pages/Students/Index.cshtml.cs` with the following code to add sorting.

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;
    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder)
    {
        // using System;
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Requires adding `using System;`.
- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL and query string is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either `Name` or `Date`. The `sortOrder` parameter is optionally followed by `_desc` to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the `default` in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
C#
```

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator, it takes three operands. The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to `name_desc`. If `sortOrder` is ***not*** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

Expand table

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

C#

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

switch (sortOrder)
{
    case "name_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        studentsIQ = studentsIQ.OrderBy(s => s.LastName);
        break;
}

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

C#

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in `Students/Index.cshtml`, with the following code. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
```

```
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in `Students/Index.cshtml.cs` with the following code to add filtering:

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }
```

```
    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder, string searchString)
    {
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        CurrentFilter = searchString;

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;
        if (!String.IsNullOrEmpty(searchString))
        {
            studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                   ||
s.FirstMidName.Contains(searchString));
        }

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the `CurrentFilter` property. The search string value is received from a text box that's added in the next section.
- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is

executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the Where method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
C#

Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in `Pages/Students/Index.cshtml` to add a **Search** button.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
```

```
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

```
browser-address-bar
```

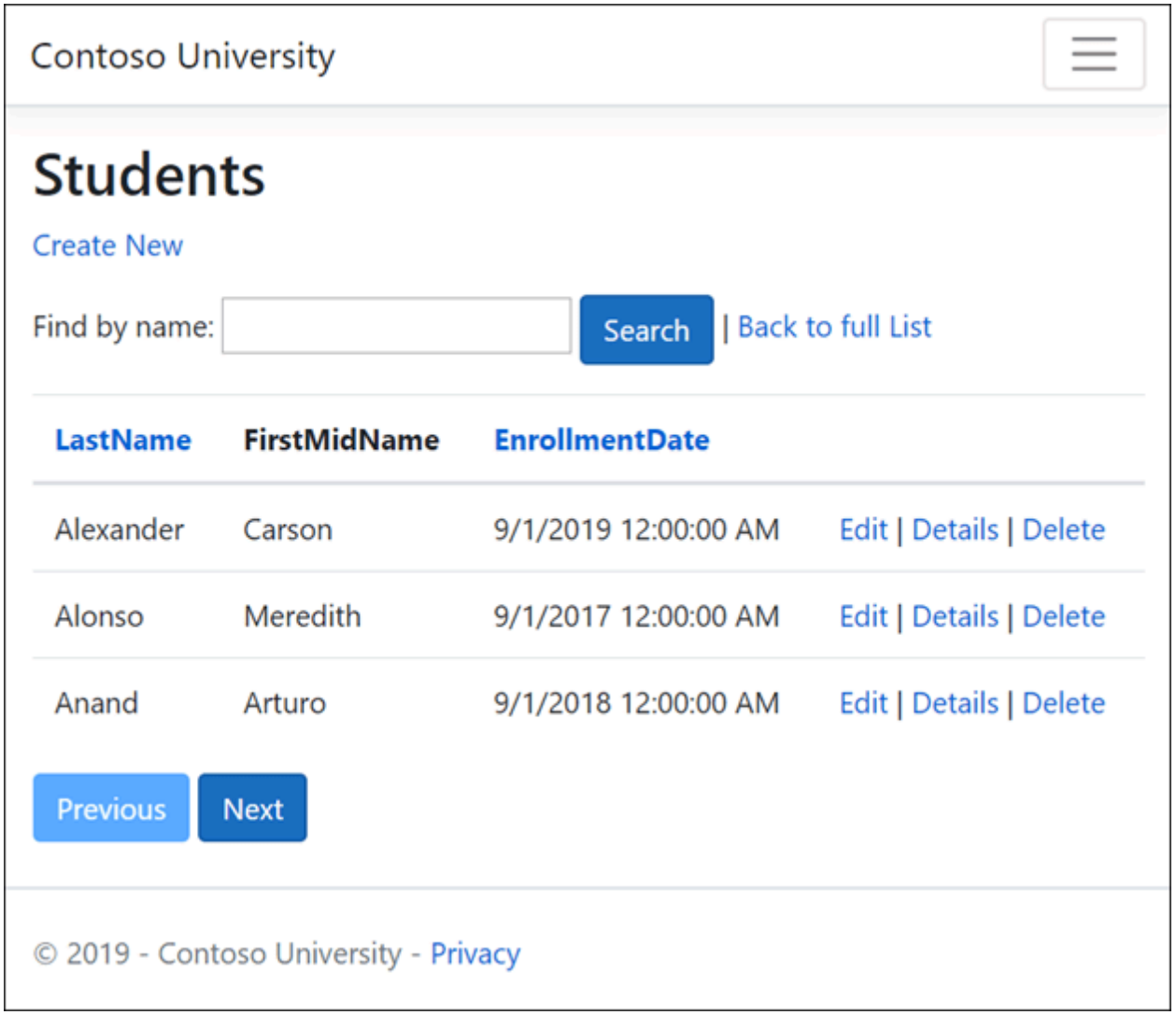
```
https://localhost:5001/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

C#

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

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
```

```
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int
pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage => PageIndex > 1;

        public bool HasNextPage => PageIndex < TotalPages;

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add page size to configuration

Add `PageSize` to the `appsettings.json` Configuration file:

JSON

```
{
  "PageSize": 3,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=CU-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Add paging to IndexModel

Replace the code in `Students/Index.cshtml.cs` to add paging.

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;
        private readonly IConfiguration Configuration;

        public IndexModel(SchoolContext context, IConfiguration
configuration)
        {
            _context = context;
            Configuration = configuration;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }
```

```
        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                       ||
s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            var pageSize = Configuration.GetValue("PageSize", 4);
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the `CurrentSort` property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.
- Sets `pageSize` to 3 from Configuration, 4 if configuration fails.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `pageIndex ?? 1` returns the value of `pageIndex` if it has a value, otherwise, it returns 1.

Add paging links

Replace the code in `Students/Index.cshtml` with the following code. The changes are highlighted:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
```

```
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.Students.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Students.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

CSHTML

```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

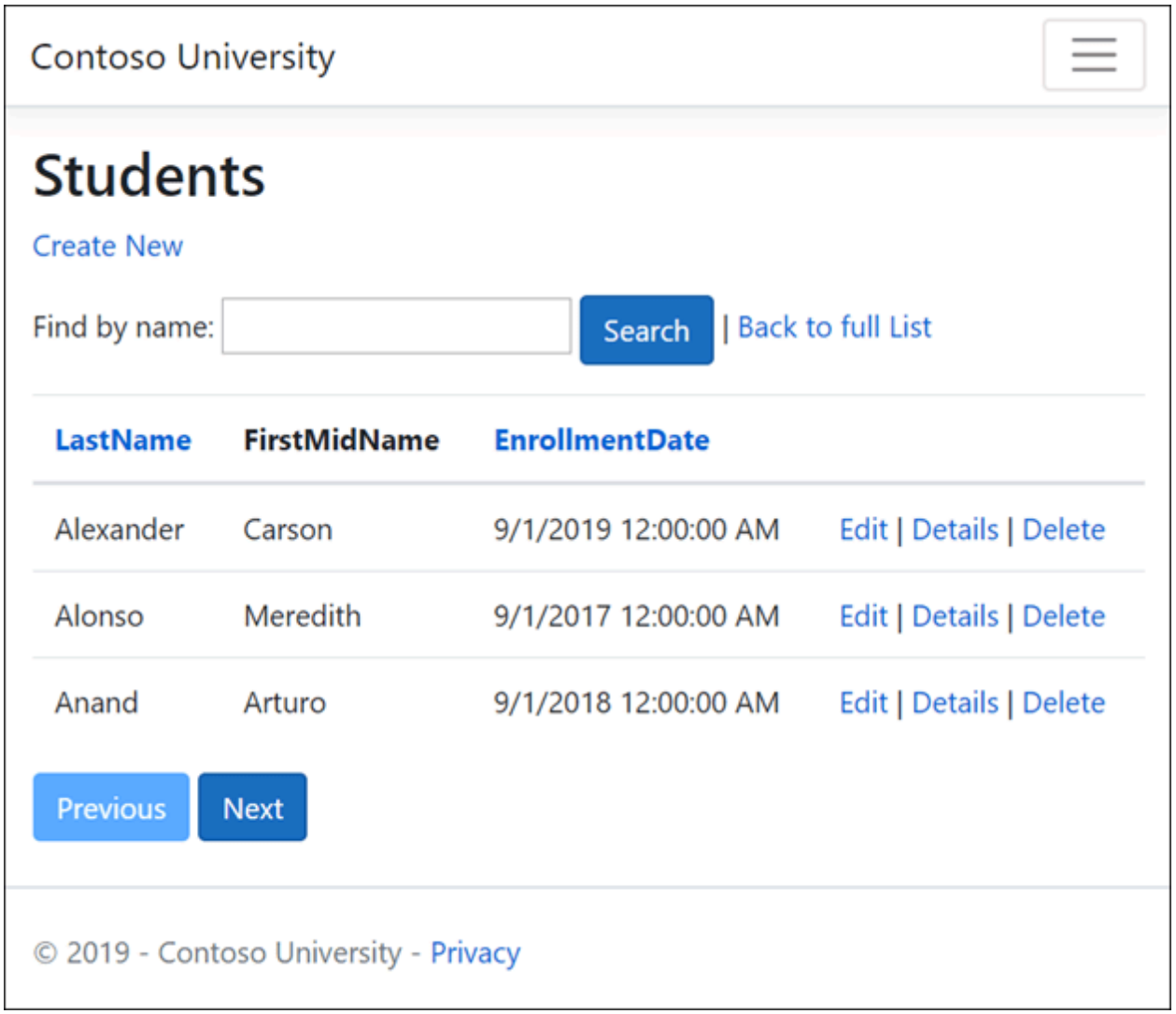
The paging buttons are displayed by tag helpers:

CSHTML

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Grouping

This section creates an `About` page that displays how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the `About` page.
- Update the `About` page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create `SchoolViewModels/EnrollmentDateGroup.cs` with the following code:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
```

```
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Create the Razor Page

Create a `Pages/About.cshtml` file with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Create the page model

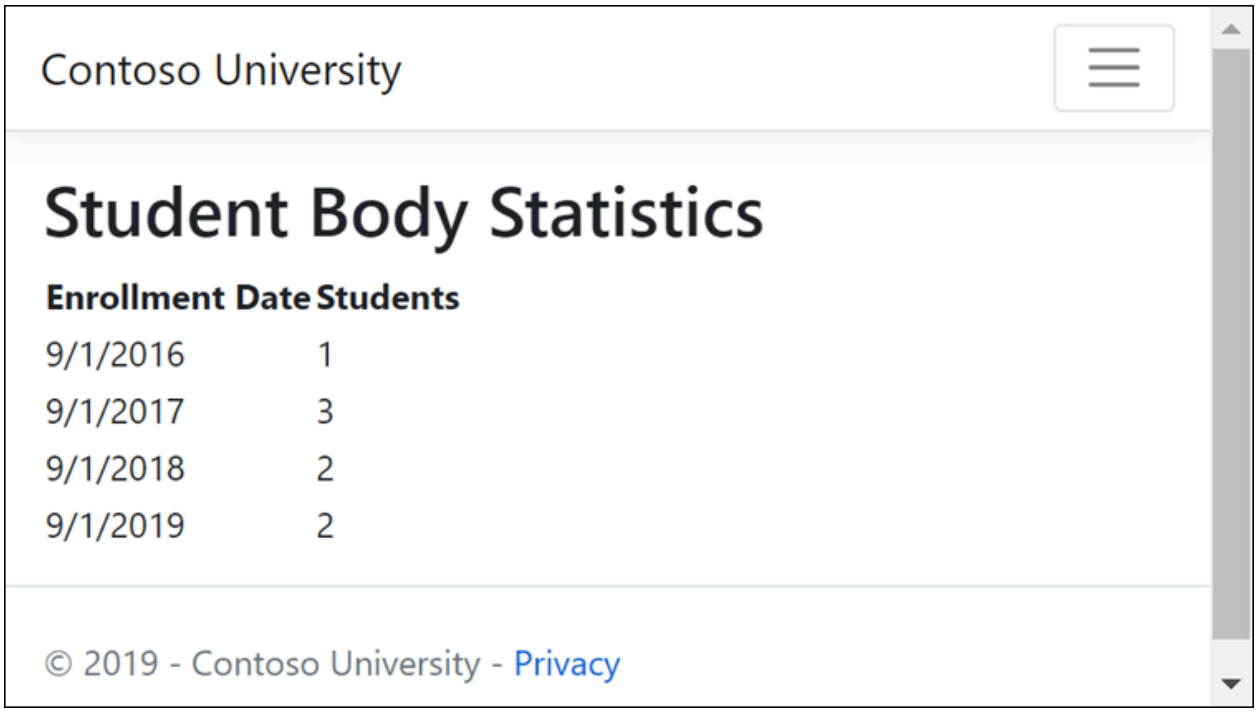
Update the `Pages/About.cshtml.cs` file with the following code:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



Next steps

In the next tutorial, the app uses migrations to update the data model.

Previous tutorial | Next tutorial

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

Article • 05/31/2024

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, the database needs to be dropped. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the DB in sync with the data model works well until the app needs to be deployed to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

> **Note**
>
> **SQLite limitations**
>
> This tutorial uses the Entity Framework Core migrations feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding

a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.

The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code goes in the `Up` and `Down` methods for a migration and involves:

- Creating a new table.
- Copying data from the old table to the new table.
- Dropping the old table.
- Renaming the new table.

Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:

- SQLite EF Core Database Provider Limitations
- Customize migration code
- Data seeding
- SQLite ALTER TABLE statement

Drop the database

Visual Studio

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database
```

Create an initial migration

Visual Studio

Run the following commands in the PMC:

PowerShell

```
Add-Migration InitialCreate
Update-Database
```

Remove EnsureCreated

This tutorial series started by using EnsureCreated. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In `Program.cs`, delete the following line:

C#

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the `Migrations\<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

C#

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
```

```
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
```

```
                table: "Enrollment",
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter (`InitialCreate` in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in `Migrations/SchoolContextModelSnapshot.cs`. When add a migration is added, EF determines what changed by comparing the current data model to the snapshot file.

Because the snapshot file tracks the state of the data model, a migration cannot be deleted by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, use the migrations remove command. `migrations remove` deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

See Resetting all migrations to remove all migrations.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```text
SqlException: Cannot open database "ContosoUniversity" requested by the
login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- dotnet ef migrations CLI commands

- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

Previous tutorial | Next tutorial

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

Article • 04/10/2024

By Tom Dykstra, Jeremy Likness, and Jon P Smith

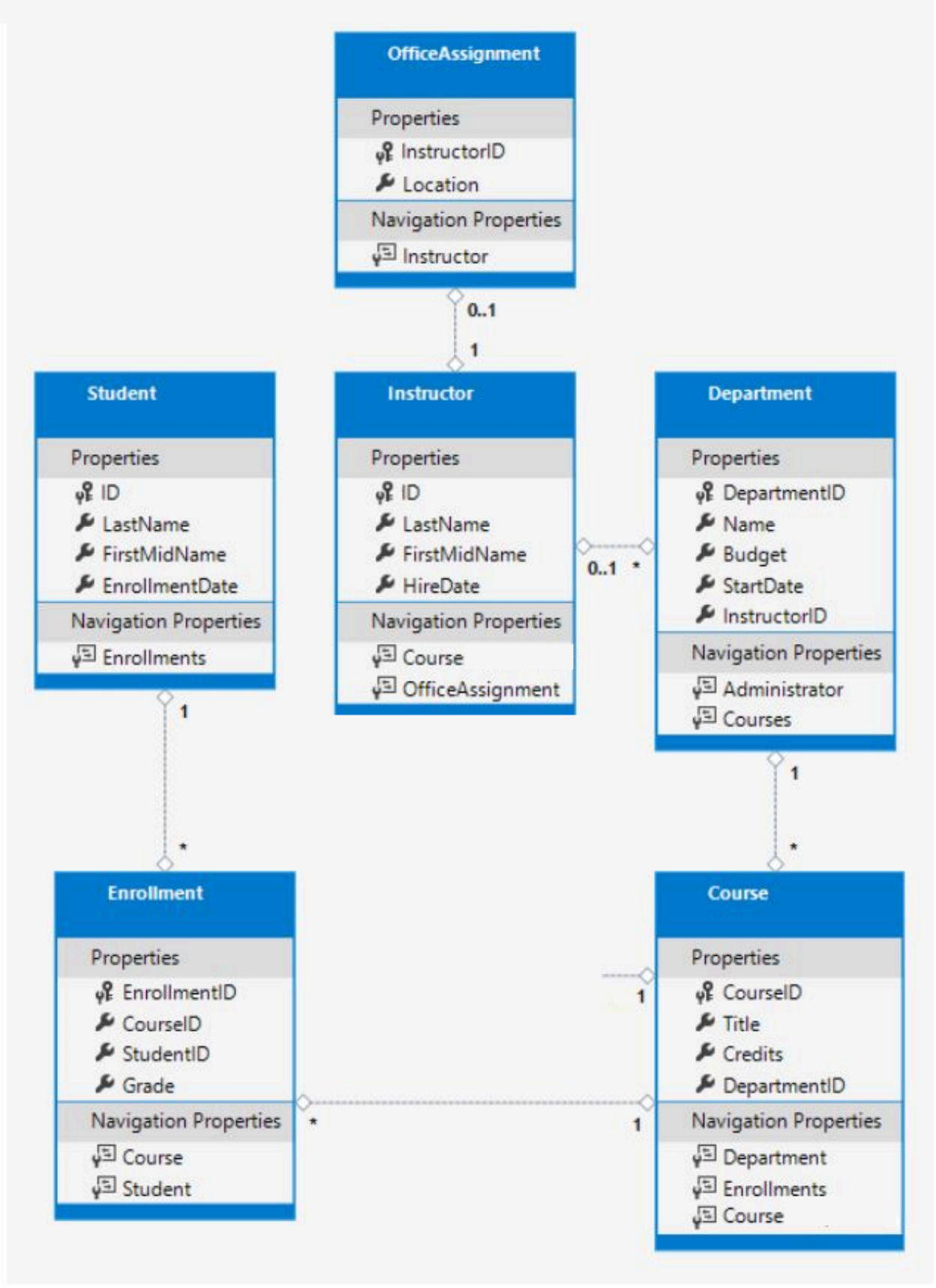
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

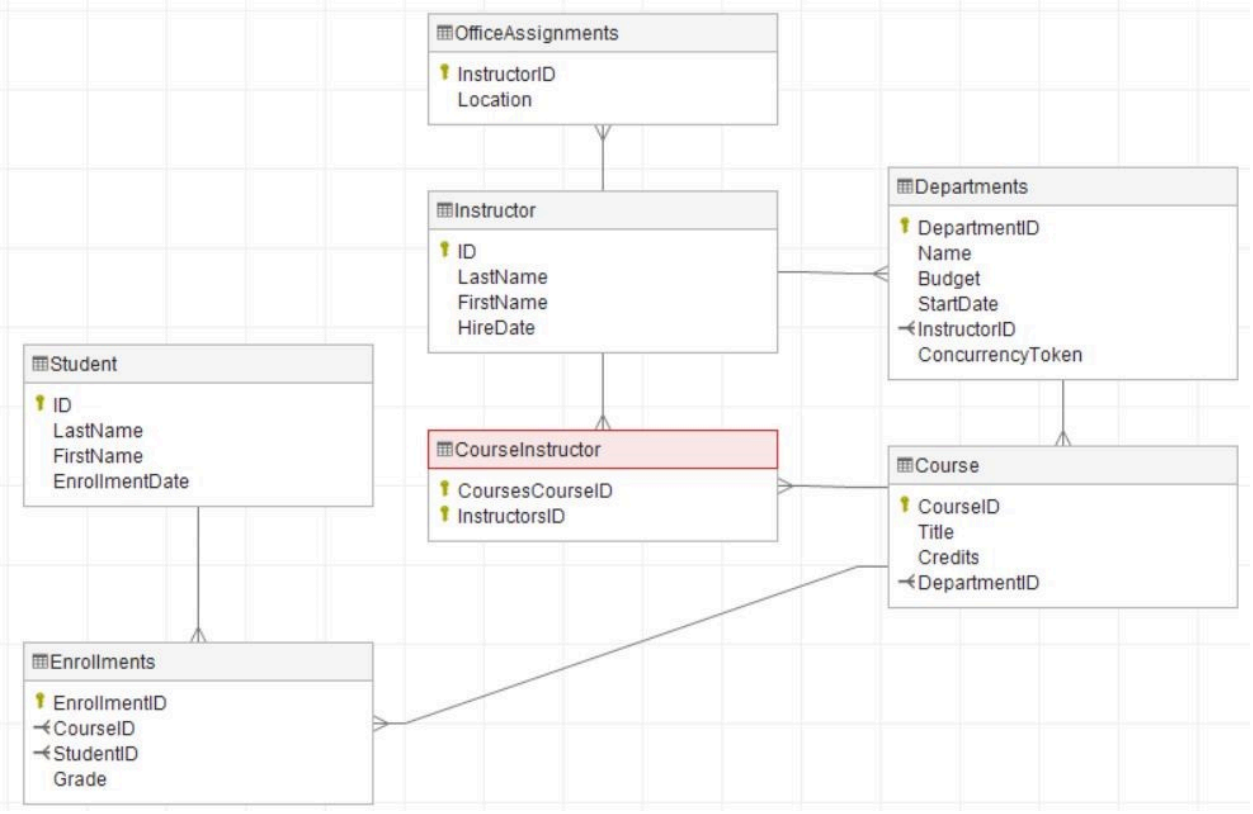
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The following database diagram was made with Dataedo:



To create a database diagram with Dataedo:

- Deploy the app to Azure
- Download and install Dataedo on your computer.
- Follow the instructions Generate documentation for Azure SQL Database in 5 minutes

In the preceding Dataedo diagram, the `CourseInstructor` is a join table created by Entity Framework. For more information, see Many-to-many

The Student entity

Replace the code in `Models/Student.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
```

```
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code adds a `FullName` property and adds the following attributes to existing properties:

- [DataType]
- [DisplayFormat]
- [StringLength]
- [Column]
- [Required]
- [Display]

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

```
C#
```

```
[DataType(DataType.Date)]
```

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
```

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

C#

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

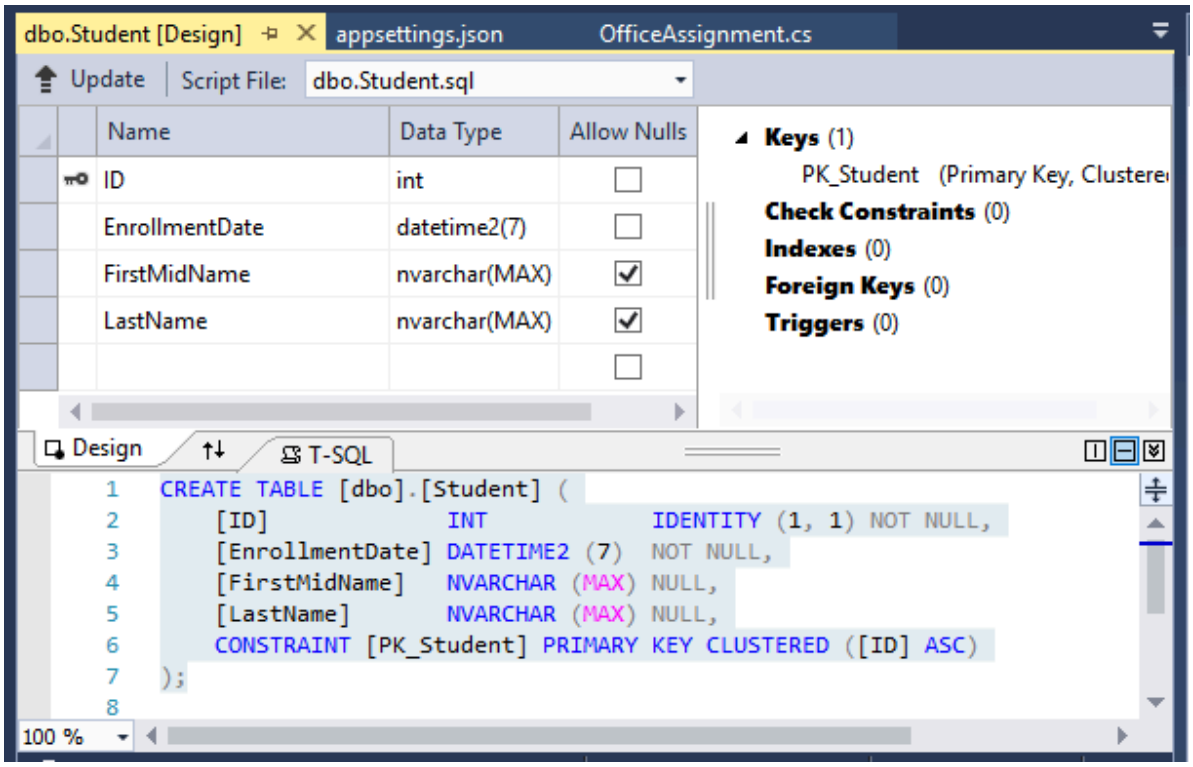
The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Visual Studio

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

C#

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
C#

[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
C#

[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
C#

[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

Visual Studio

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
There are pending model changes
Pending model changes are detected in the following:

SchoolContext
```

- In the PMC, enter the following commands to create a new migration and update the database:

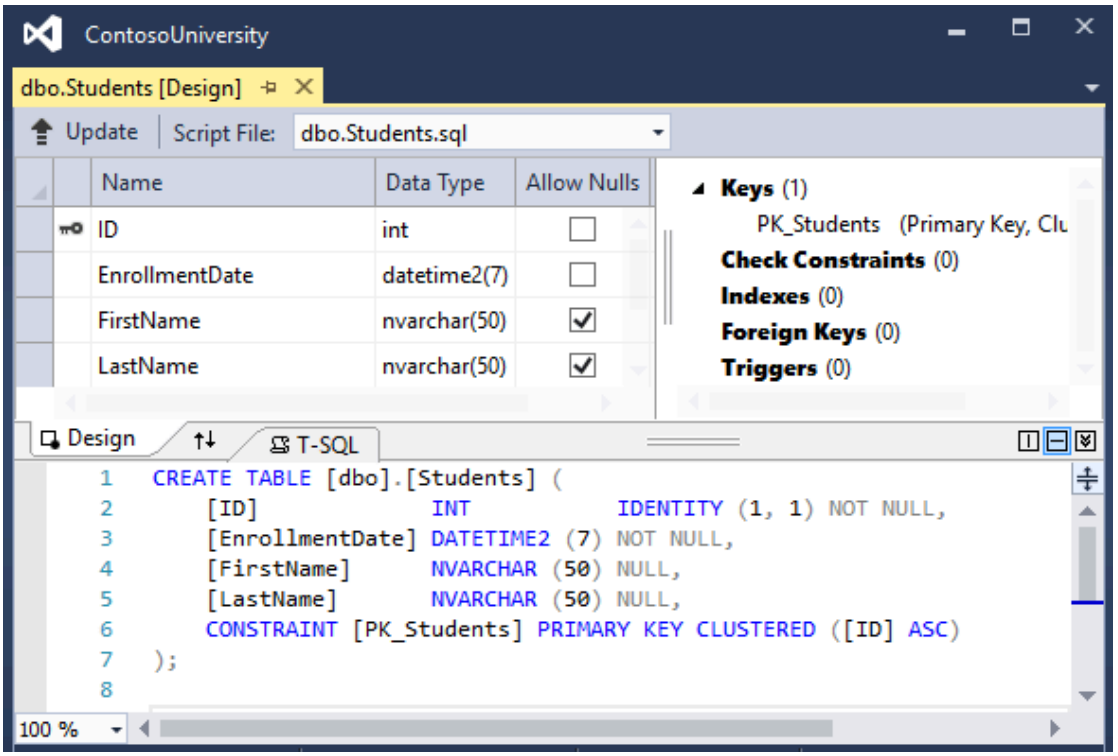
  ```PowerShell
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

  ```text
  An operation was scaffolded that may result in the loss of data.
  Please review the migration for accuracy.
  ```

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:



Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> ⓘ Note
>
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity

Create `Models/Instructor.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
```

```
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<Course> Courses { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

C#

```
[DataType(DataType.Date),Display(Name = "Hire
Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

Navigation properties

The `Courses` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `Courses` is defined as a collection.

C#

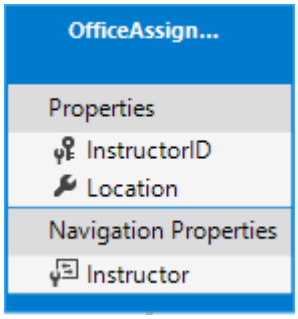
```
public ICollection<Course> Courses { get; set; }
```

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

C#

```
public OfficeAssignment OfficeAssignment { get; set; }
```

The OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The [Key] attribute is used to identify a property as the primary key (PK) when the property name is something other than `classnameID` or `ID`.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity. A one-to-zero-or-one relationship occurs when a PK in one table is both a PK and a FK in another table.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

C#

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship. For more information, see EF Keys.

The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity

Update `Models/Course.cs` with the following code:

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```