A model is a set of classes that represent the data that the app manages. The model for this app is a single TodoItem class.

- In Solution Explorer, right-click the project. Select Add > New Folder. Name the folder Models.
- Right-click the Models folder and select Add > Class. Name the class TodoItem and select Add.
- Replace the template code with the following code:

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

The Id property functions as the unique key in a relational database.
Model classes can go anywhere in the project, but the Models folder is used by convention.

# Add a database context

The database context is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

## Add Microsoft.EntityFrameworkCore.SqlServer

- From the Tools menu, select NuGet Package Manager > Manage NuGet Packages for Solution.
- Select the Browse tab, and then enter Microsoft.EntityFrameworkCore.SqlServer in the search box.
- Select Microsoft.EntityFrameworkCore.SqlServer in the left pane.
- Select the Project check box in the right pane and then select Install.
- Use the preceding instructions to add the Microsoft.EntityFrameworkCore.InMemory NuGet package.

# Add the TodoContext database context

- Right-click the Models folder and select Add > Class. Name the class TodoContext and click Add.

- Enter the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

## Register the database context

In ASP.NET Core, services such as the DB context must be registered with the [dependency injection (DI)] container. The container provides the service to controllers.
Update Startup.cs with the following highlighted code:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
```

```
            services.AddDbContext<TodoContext>(opt =>
               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

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

The preceding code:

- Removes unused using declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

## Scaffold a controller

- Right-click the Controllers folder.
- Select Add > New Scaffolded Item.
- Select API Controller with actions, using Entity Framework, and then select Add.
- In the Add API Controller with actions, using Entity Framework dialog:
    - Select TodoItem (TodoApi.Models) in the Model class.
    - Select TodoContext (TodoApi.Models) in the Data context class.
    - Select Add.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (TodoContext) into the controller. The database context is used in each of the CRUD methods in the controller.

# Examine the PostTodoItem create method

Replace the return statement in the PostTodoItem to use the nameof operator:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

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

The preceding code is an HTTP POST method, as indicated by the [HttpPost] attribute. The method gets the value of the to-do item from the body of the HTTP request.
The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the GetTodoItem action to create the Location header's URI. The C# nameof keyword is used to avoid hard-coding the action name in the CreatedAtAction call.

## Install Postman

This tutorial uses Postman to test the web API.

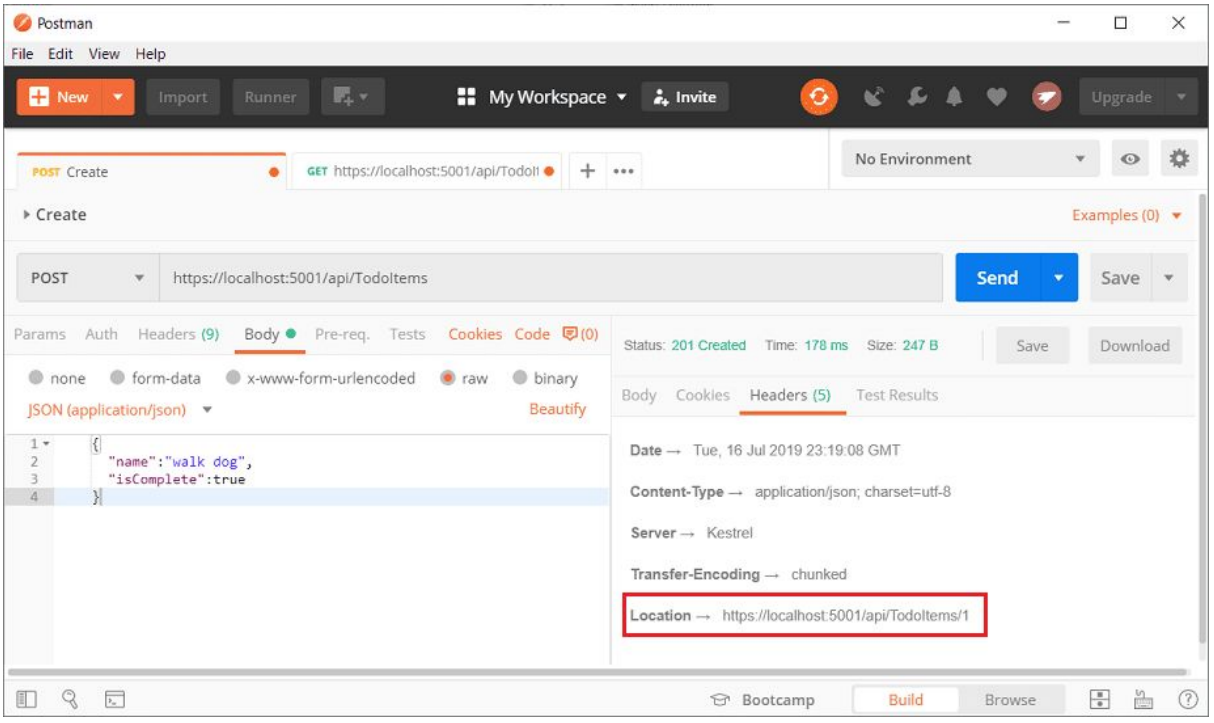
- Install Postman
- Start the web app.
- Start Postman.
- Disable SSL certificate verification
    - From File > Settings (General tab), disable SSL certificate verification.
      Warning
      Re-enable SSL certificate verification after testing the controller.

## Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to POST.
- Select the Body tab.
- Select the raw radio button.
- Set the type to JSON (application/json).
- In the request body enter JSON for a to-do item:

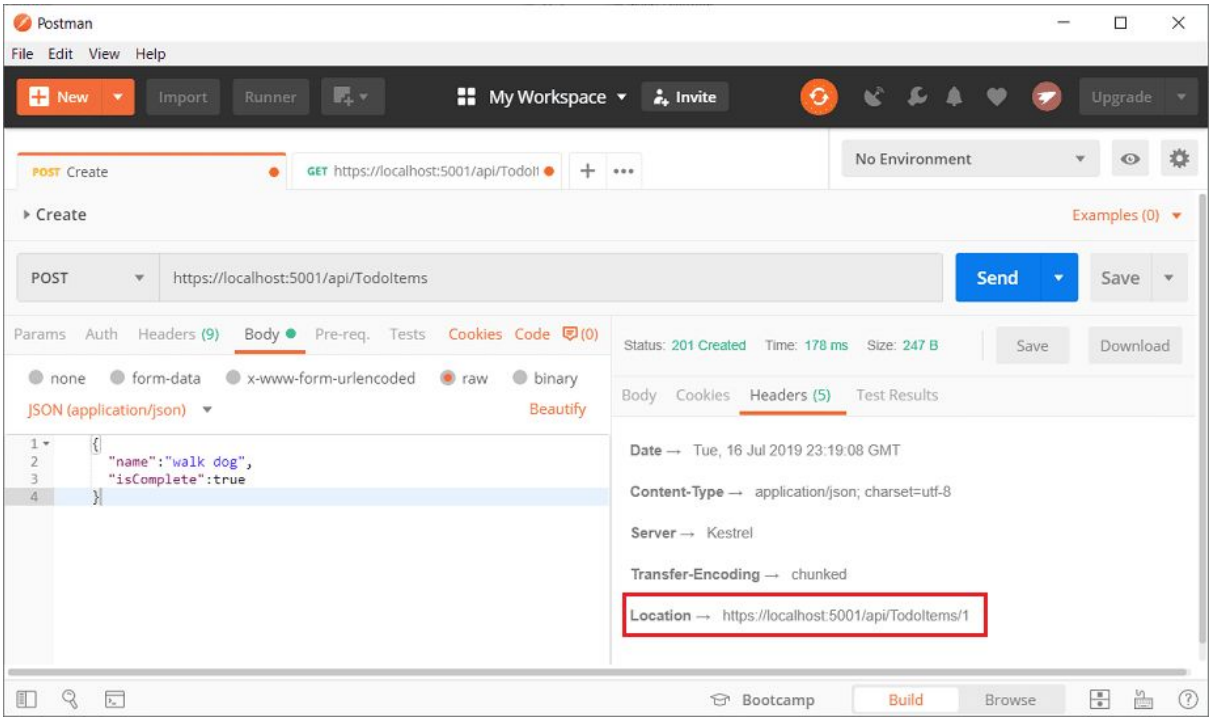
```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select Send.




## Test the location header URI

- Select the Headers tab in the Response pane.
- Copy the Location header value:



- Set the method to GET.
- Paste the URI (for example, https://localhost:5001/api/TodoItems/1).
- Select Send.

## Examine the GET methods

These methods implement two GET endpoints:

- GET /api/TodoItems
- GET /api/TodoItems/{id}

Test the app by calling the two endpoints from a browser or Postman. For example:

- https://localhost:5001/api/TodoItems
- https://localhost:5001/api/TodoItems/1

A response similar to the following is produced by the call to GetTodoItems:

```
JSON
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

### Test Get with Postman

- Create a new request.
- Set the HTTP method to GET.
- Set the request URL to https://localhost:<port>/api/TodoItems. For example, https://localhost:5001/api/TodoItems.
- Set Two pane view in Postman.
- Select Send.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not 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:

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

```
public TodoItemsController(TodoContext context)
{
    _context = context;
}
```

- 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 TodoItemsController, 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.

```
// GET: api/TodoItems/5
[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, 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 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:

```
// PUT: api/TodoItems/5
[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.
If you get an error calling PutTodoItem, call GET to ensure there's an item in the database.

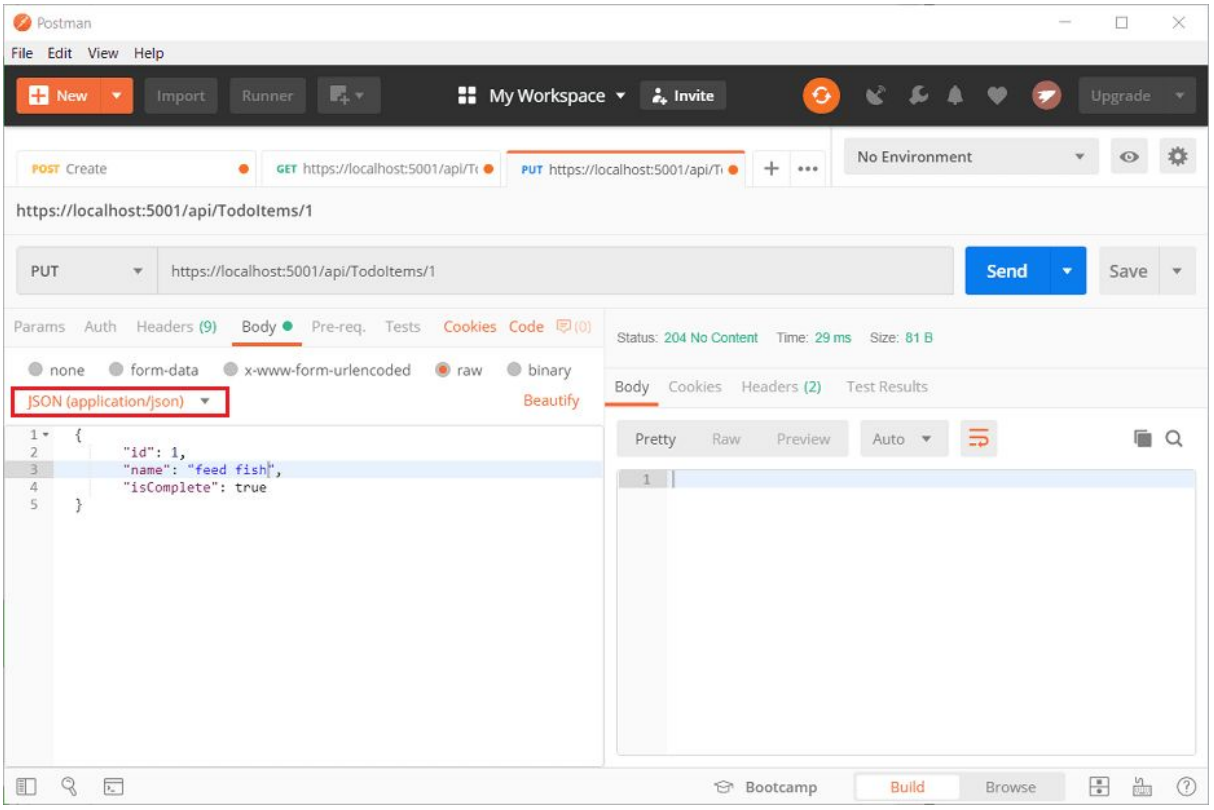
## Test the PutTodoItem method

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

```
 {
    "ID":1,
    "name":"feed fish",
    "isComplete":true
```

}

The following image shows the Postman update:




## The DeleteTodoItem method

Examine the DeleteTodoItem method:

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

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

    return todoItem;
}
```

## Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to DELETE.
- Set the URI of the object to delete (for example https://localhost:5001/api/TodoItems/1).
- Select Send.