

Учебник. Создание веб-API с помощью ASP.NET Core

25.02.2020 • Время чтения: 25 мин •   

В этой статье

[Обзор](#)

[Предварительные требования](#)

[Создайте веб-проект.](#)

[Добавление класса модели](#)

[Добавление контекста базы данных](#)

[Добавление контекста базы данных TodoContext](#)

[Регистрация контекста базы данных](#)

[Формирование шаблонов контроллера](#)

[Знакомство с методом создания PostTodoItem](#)

[Знакомство с методами GET](#)

[Маршрутизация и пути URL](#)

[Возвращаемые значения](#)

[Метод PutTodoItem](#)

[Метод DeleteTodoItem](#)

[Предотвращение избыточной публикации](#)

[Вызов веб-API с помощью JavaScript](#)

[Добавление поддержки аутентификации в веб-API](#)

[Дополнительные ресурсы](#)

Авторы: [Рик Андерсон](#) (Rick Anderson), [Кирк Ларкин](#) (Kirk Larkin) и [Майк Уоссон](#) (Mike Wasson)

В этом учебнике приводятся основные сведения о создании веб-API с помощью ASP.NET Core.

В этом руководстве вы узнаете, как:

- ✓ Создание проекта веб-API.
- ✓ Добавление класса модели и контекста базы данных.
- ✓ Формирование шаблонов контроллера с использованием методов CRUD.
- ✓ Настройка маршрутизации, URL-пути и возвращаемых значений.
- ✓ Вызов веб-API с помощью Postman.

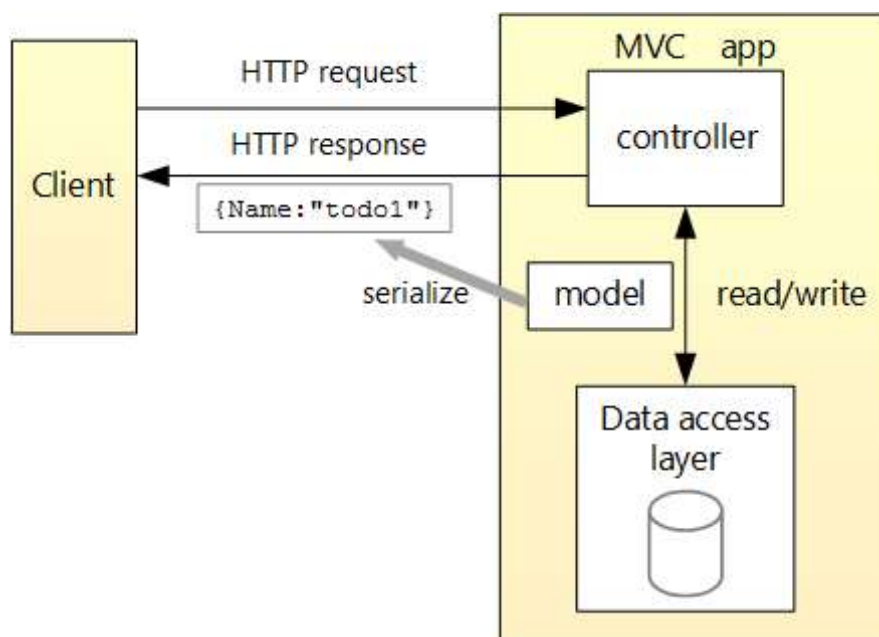
В итоге вы получите веб-API, позволяющий работать с элементами списка дел, хранимыми в базе данных.

Обзор

В этом руководстве создается следующий API-интерфейс:

API	Описание	Текст запроса	Текст ответа
GET /api/TodoItems	Получение всех элементов задач	Отсутствуют	Массив элементов задач
GET /api/TodoItems/{id}	Получение объекта по идентификатору	Отсутствуют	Элемент задачи
POST /api/TodoItems	Добавление нового элемента	Элемент задачи	Элемент задачи
PUT /api/TodoItems/{id}	Обновление существующего элемента	Элемент задачи	Отсутствуют
DELETE /api/TodoItems/{id}	Удаление элемента	Отсутствуют	Отсутствуют

На следующем рисунке показана структура приложения.



Предварительные требования

Visual Studio

Visual Studio Code

Visual Studio для Mac

- [Visual Studio 2019 16.4 или более поздней версии](#) с рабочей нагрузкой **ASP.NET** и разработка веб-приложений
- [Пакет SDK для .NET Core 3.1 или более поздней версии](#)

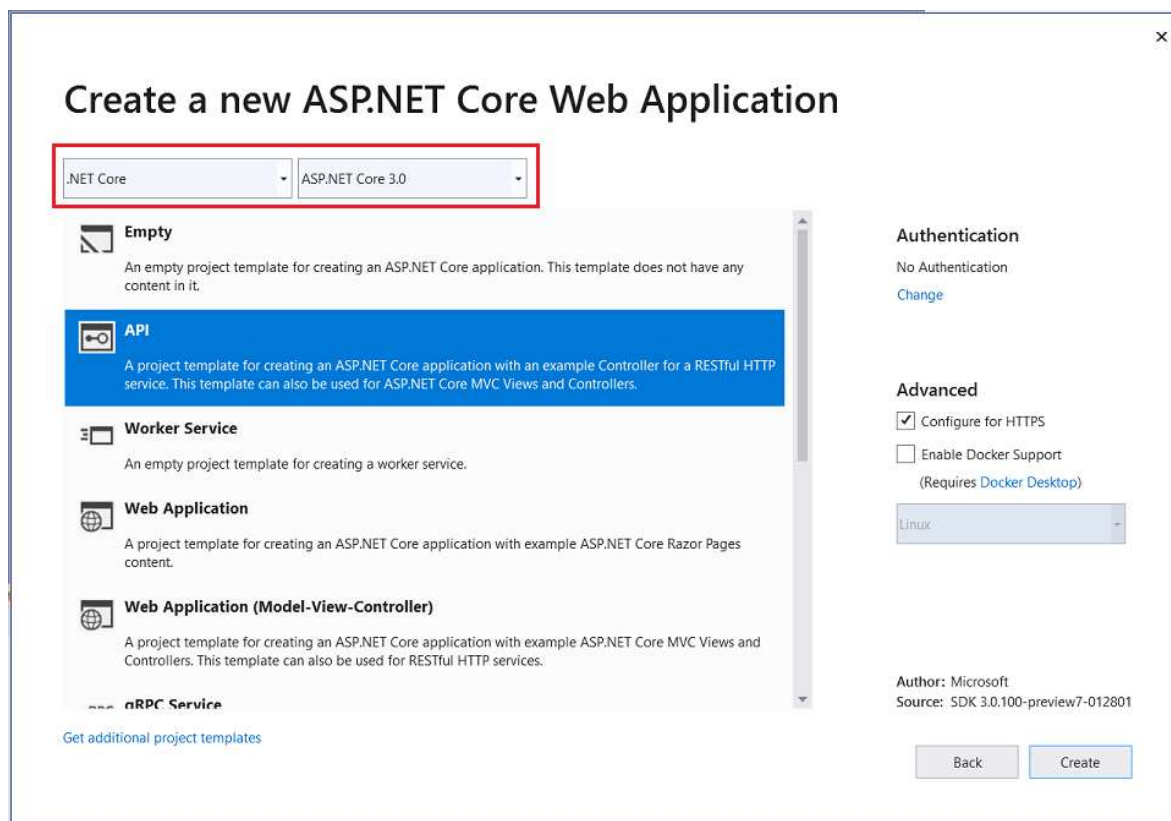
Создайте веб-проект.

Visual Studio

Visual Studio Code

Visual Studio для Mac

- В меню **Файл** выберите пункт **Создать > Проект**.
- Выберите шаблон **Веб-приложение ASP.NET Core** и нажмите **Далее**.
- Назовите проект *ToDoApi* и нажмите **Создать**.
- В диалоговом окне **Создание веб-приложения ASP.NET Core** убедитесь в том, что выбраны платформы **.NET Core** и **ASP.NET Core 3.1**. Выберите шаблон **API** и нажмите кнопку **Создать**.



Тестирование API

Шаблон проекта создает API weatherForecast. Вызовите метод get из браузера для тестирования приложения.

Visual Studio

Visual Studio Code

Visual Studio для Mac

Нажмите клавиши CTRL+F5, чтобы запустить приложение. Visual Studio запустит браузер и перейдет к `https://localhost:<port>/WeatherForecast`, где `<port>` — это номер порта, выбранный случайным образом.

Если появится диалоговое окно с запросом о необходимости доверять сертификату IIS Express, выберите **Да**. В появляющемся следом диалоговом окне **Предупреждение системы безопасности** выберите **Да**.

Возвращаемые данные JSON будут выглядеть примерно так:

JSON

 Копировать

```
[
  {
    "date": "2019-07-16T19:04:05.7257911-06:00",
    "temperatureC": 52,
    "temperatureF": 125,
    "summary": "Mild"
  },
  {
    "date": "2019-07-17T19:04:05.7258461-06:00",
    "temperatureC": 36,
    "temperatureF": 96,
    "summary": "Warm"
  },
  {
    "date": "2019-07-18T19:04:05.7258467-06:00",
    "temperatureC": 39,
    "temperatureF": 102,
    "summary": "Cool"
  },
  {
    "date": "2019-07-19T19:04:05.7258471-06:00",
    "temperatureC": 10,
    "temperatureF": 49,
    "summary": "Bracing"
  },
  {
    "date": "2019-07-20T19:04:05.7258474-06:00",
    "temperatureC": -1,
    "temperatureF": 31,
    "summary": "Chilly"
  }
]
```

Добавление класса модели

Модель — это набор классов, представляющих данные, которыми управляет приложение. Модель этого приложения содержит единственный класс `TodoItem`.

Visual Studio

Visual Studio Code

Visual Studio для Mac

- В **обозревателе решений** щелкните проект правой кнопкой мыши. Выберите **Добавить > Новая папка**. Присвойте папке имя *Models*.
- Щелкните папку *Models* правой кнопкой мыши и выберите **Добавить > Класс**. Присвойте классу имя *TodoItem* и выберите **Добавить**.
- Замените код шаблона следующим кодом:

C#

 Копировать

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

Свойство `Id` выступает в качестве уникального ключа реляционной базы данных.

Классы моделей можно размещать в любом месте проекта, но обычно для этого используется папка *Models*.

Добавление контекста базы данных

Контекст базы данных — это основной класс, который координирует функциональные возможности Entity Framework для модели данных. Этот класс является производным от класса `Microsoft.EntityFrameworkCore.DbContext`.

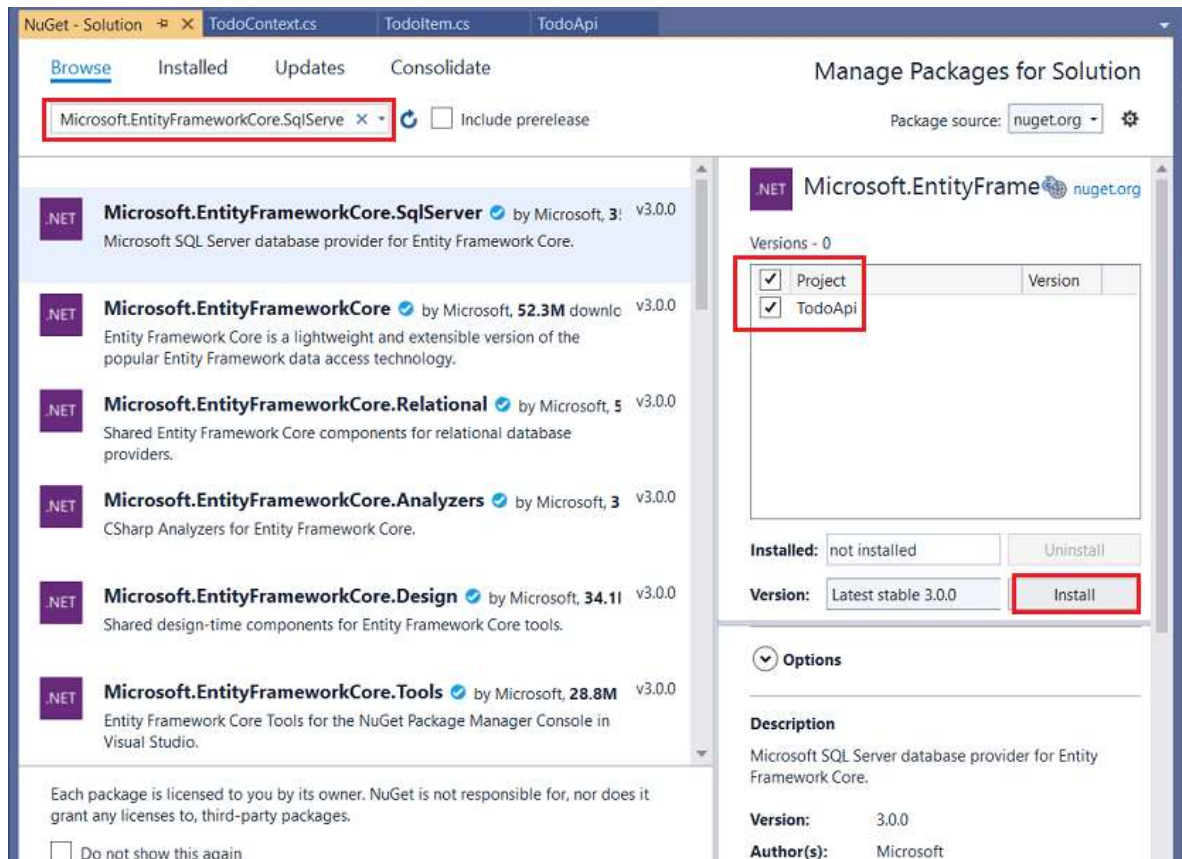
Visual Studio

Visual Studio Code/Visual Studio для Mac

Добавление `Microsoft.EntityFrameworkCore.SqlServer`

- В меню **Сервис** выберите **Диспетчер пакетов NuGet > Управление пакетами NuGet для решения**.

- Перейдите на вкладку **Обзор** и введите **Microsoft.EntityFrameworkCore.SqlServer** в поле поиска.
- На панели слева выберите **Microsoft.EntityFrameworkCore.SqlServer**.
- Установите флажок **Проект** на правой панели и выберите **Установить**.
- Добавьте пакет NuGet Microsoft.EntityFrameworkCore.InMemory согласно инструкциям выше.



Добавление контекста базы данных TodoContext

- Щелкните папку *Models* правой кнопкой мыши и выберите **Добавить > Класс**. Назовите класс *TodoContext* и нажмите **Добавить**.

- Введите следующий код:

C#

Копировать

```

using Microsoft.EntityFrameworkCore;

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

```

```
        : base(options)
        {
        }

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

Регистрация контекста базы данных

В ASP.NET Core службы (такие как контекст базы данных) должны быть зарегистрированы с помощью контейнера [внедрения зависимостей](#). Контейнер предоставляет службу контроллерам.

Обновите файл *Startup.cs*, используя следующий выделенный код:

C#

 Копировать

```
// Unused usings removed
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();
    });
}
}
}
}
}

```

Предыдущий код:

- Удалите неиспользуемые объявления `using`.
- Добавляет контекст базы данных в контейнер внедрения зависимостей.
- Указывает, что контекст базы данных будет использовать базу данных в памяти.

Формирование шаблонов контроллера

Visual Studio

Visual Studio Code/Visual Studio для Mac

- Щелкните папку *Controllers* правой кнопкой мыши.
- Щелкните **Добавить > Создать шаблонный элемент**.
- Выберите **Контроллер API с действиями, использующий Entity Framework**, а затем выберите **Добавить**.
- В диалоговом окне **Контроллер API с действиями, использующий Entity Framework** сделайте следующее:
 - Выберите **TodoItem (TodoApi.Models)** в поле **Класс модели**.
 - Выберите **TodoContext (TodoApi.Models)** в поле **Класс контекста данных**.
 - Нажмите **Добавить**.

Сформированный код:

- Пометьте этот класс атрибутом [\[ApiController\]](#). Этот атрибут указывает, что контроллер отвечает на запросы веб-API. Дополнительные сведения о поведении, которое реализует этот атрибут, см. в [Создание веб-API с помощью ASP.NET Core](#).
- Использует внедрение зависимостей для внедрения контекста базы данных (TodoContext) в контроллер. Контекст базы данных используется в каждом методе [создания, чтения, обновления и удаления](#) в контроллере.


Шаблоны ASP.NET Core для:

- Контроллеры с представлениями включают `[action]` в шаблоне маршрута.
- Контроллеры API не включают `[action]` в шаблоне маршрута.

Если токен `[action]` не находится в шаблоне маршрута, имя [действия](#) исключается из маршрута. То есть имя связанного метода действия не используется в соответствующем маршруте.

Знакомство с методом создания PostTodoItem

Измените инструкцию возврата в `PostTodoItem` и используйте оператор [nameof](#):

C#	 Копировать
<pre>// 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); }</pre>	

Предыдущий код является методом HTTP POST, обозначенным атрибутом [\[HttpPost\]](#). Этот метод получает значение элемента списка дел из текста HTTP-запроса.

Метод [CreatedAtAction](#):

- В случае успеха возвращает код состояния HTTP 201. HTTP 201 представляет собой стандартный ответ для метода HTTP POST, создающий ресурс на сервере.

- Добавляет в ответ заголовок [Location](#). Заголовок Location указывает [URI](#) новой созданной задачи. Дополнительные сведения см. в статье [10.2.2 201 "Создан ресурс"](#).
- Указывает действие `GetTodoItem` для создания URI заголовка Location. Ключевое слово `nameof` C# используется для предотвращения жесткого программирования имени действия в вызове `CreatedAtAction`.

Установка Postman

В этом учебнике для тестирования веб-API используется Postman.


- Установка [Postman](#)
- Запустите веб-приложение.
- Запустите Postman.
- Отключение параметра **Проверка SSL-сертификата**
 - В меню **Файл** > **Параметры** (вкладка **Общие**), отключите параметр **Проверка SSL-сертификата**.

Предупреждение

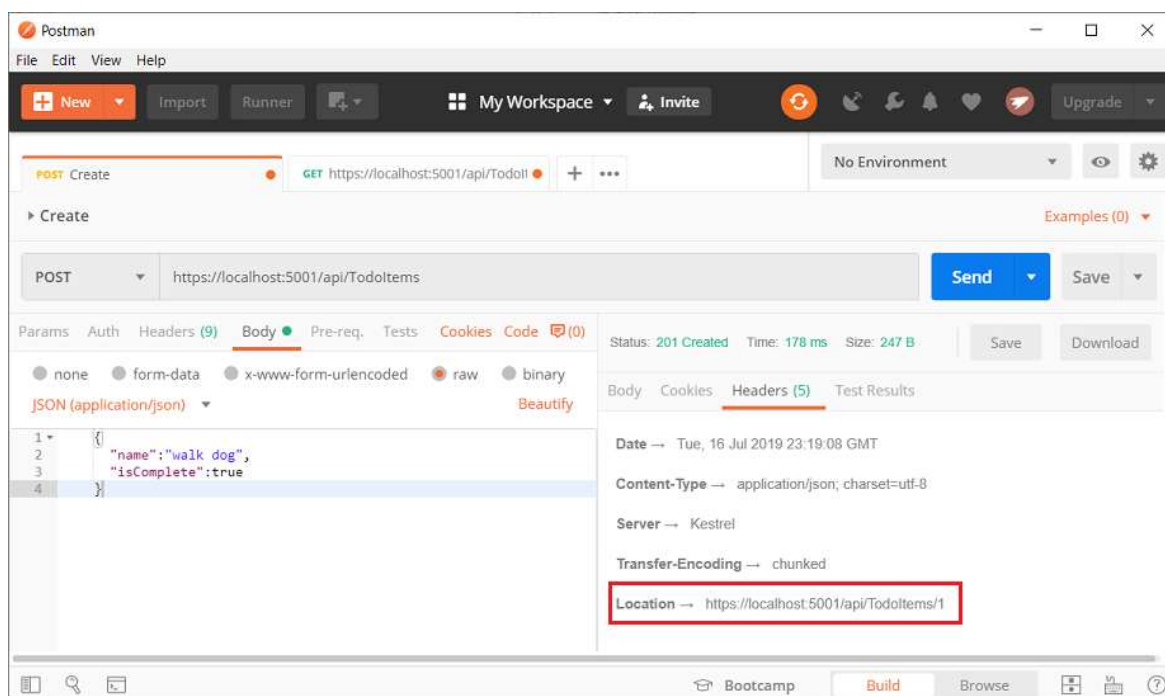
После тестирования контроллера снова включите проверку SSL-сертификата.

Тестирование `PostTodoItem` с использованием Postman

- Создайте новый запрос.
- Установите HTTP-метод `POST`.
- Откройте вкладку **Тело**.
- Установите переключатель **без обработки**.
- Задайте тип **JSON (приложение/json)**.
- В теле запроса введите код JSON для элемента списка дел:

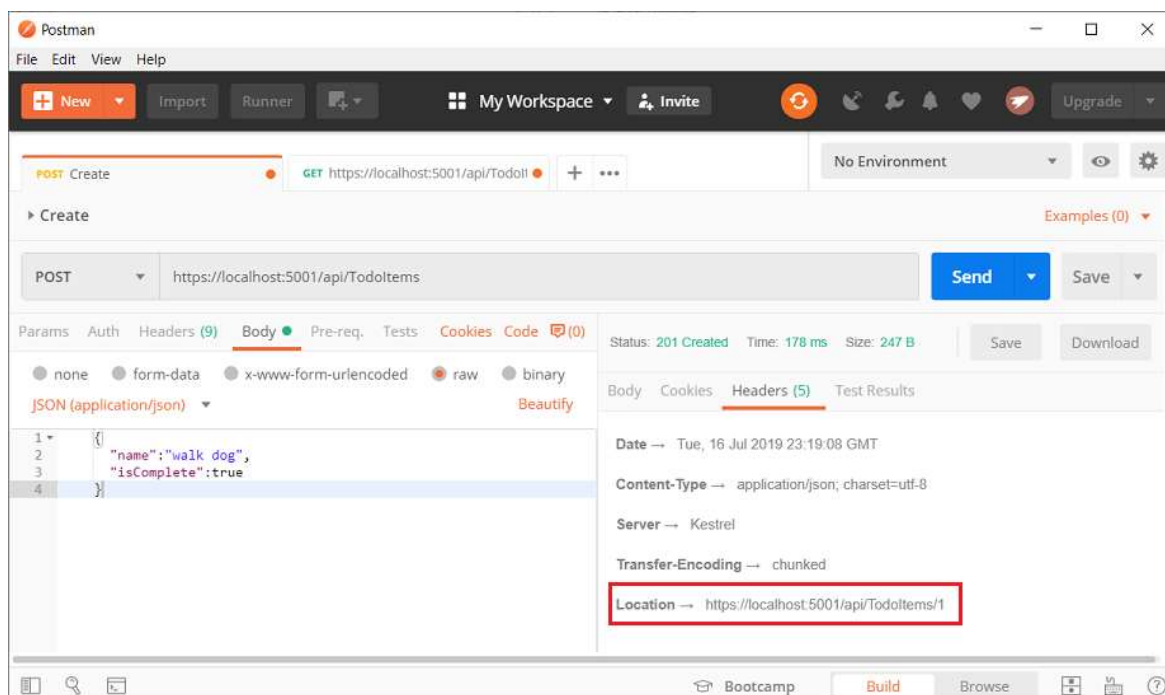
JSON	 Копировать
<pre>{ "name": "walk dog", "isComplete": true }</pre>	

- Нажмите кнопку **Отправить**.



Тестирование URI заголовка расположения

- Перейдите на вкладку **Заголовки** в области **Ответ**.
- Скопируйте значение заголовка **Расположение**:



- Укажите метод GET.
- Вставьте URI (например, https://localhost:5001/api/TodoItems/1).
- Нажмите кнопку **Отправить**.

Знакомство с методами GET


Эти методы реализуют две конечные точки GET:

- GET /api/ToDoItems
- GET /api/ToDoItems/{id}

Протестируйте приложение, вызвав эти две конечные точки в браузере или в Postman. Пример:

- <https://localhost:5001/api/ToDoItems>
- <https://localhost:5001/api/ToDoItems/1>

При вызове `GetToDoItems` возвращается примерно такой ответ:

JSON	 Копировать
<pre>[{ "id": 1, "name": "Item1", "isComplete": false }]</pre>	

Тестирование Get с использованием Postman


- Создайте новый запрос.
- Укажите метод HTTP **GET**.
- Укажите URL-адрес запроса `https://localhost:<port>/api/ToDoItems`.
Например, `https://localhost:5001/api/ToDoItems`.
- Выберите режим **Представление с двумя областями** в Postman.
- Нажмите кнопку **Отправить**.

Это приложение использует выполняющуюся в памяти базу данных. Если остановить и вновь запустить его, предшествующий запрос GET не возвратит никаких данных. Если данные не возвращаются, данные для приложения получаются методом [POST](#).

Маршрутизация и пути URL


Атрибут [\[HttpGet\]](#) обозначает метод, который отвечает на запрос HTTP GET. Путь URL для каждого метода формируется следующим образом:

- Возьмите строку шаблона в атрибуте Route контроллера:

C#	 Копировать
<pre>[Route("api/[controller]")] [ApiController] public class TodoItemsController : ControllerBase { private readonly TodoContext _context; public TodoItemsController(TodoContext context) { _context = context; } }</pre>	

- Замените [controller] именем контроллера (по соглашению это имя класса контроллера без суффикса "Controller"). В этом примере класс контроллера имеет имя **TodolItems**, а сам контроллер, соответственно, — "TodolItems". В ASP.NET Core [маршрутизация](#) реализуется без учета регистра символов.
- Если атрибут [HttpGet] имеет шаблон маршрута (например, [HttpGet("products")]), добавьте его к пути. В этом примере шаблон не используется. Дополнительные сведения см. в разделе [Маршрутизация атрибутов с помощью атрибутов Http\[Verb\]](#).

В следующем методе GetTodoItem``"{id}" — это переменная-заполнитель для уникального идентификатора элемента задачи. При вызове GetTodoItem параметру метода id присваивается значение "{id}" в URL-адресе.

C#	 Копировать
<pre>// 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; }</pre>	

Возвращаемые значения

Возвращаемое значение имеет тип `GetTodoItems`, а метод `GetTodoItem` имеет тип [ActionResult<T> type](#). ASP.NET Core автоматически сериализует объект в формат [JSON](#) и записывает данные JSON в тело сообщения ответа. Код ответа для этого типа возвращаемого значения равен 200, что свидетельствует об отсутствии необработанных исключений. Необработанные исключения преобразуются в ошибки 5xx.

Типы возвращаемых значений `ActionResult` могут представлять широкий спектр кодов состояний HTTP. Например, метод `GetTodoItem` может возвращать два разных значения состояния:

- Если запрошенному идентификатору не соответствует ни один элемент, метод возвращает ошибку 404 ([Не найдено](#)).
- В противном случае метод возвращает код 200 с телом ответа JSON. При возвращении `item` возвращается ответ HTTP 200.

Метод PutTodoItem

Проверьте метод `PutTodoItem`.

C#

 Копировать

```
// 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();  
}
```


Страница `PutTodoItem` аналогична странице `PostTodoItem`, но использует запрос HTTP PUT. Ответ — [204 \(Нет содержимого\)](#). Согласно спецификации HTTP, запрос PUT требует, чтобы клиент отправлял всю обновленную сущность, а не только изменения. Чтобы обеспечить поддержку частичных обновлений, используйте [HTTP PATCH](#).

Если возникнет ошибка вызова `PutTodoItem`, вызовите GET, чтобы в базе данных был один элемент.

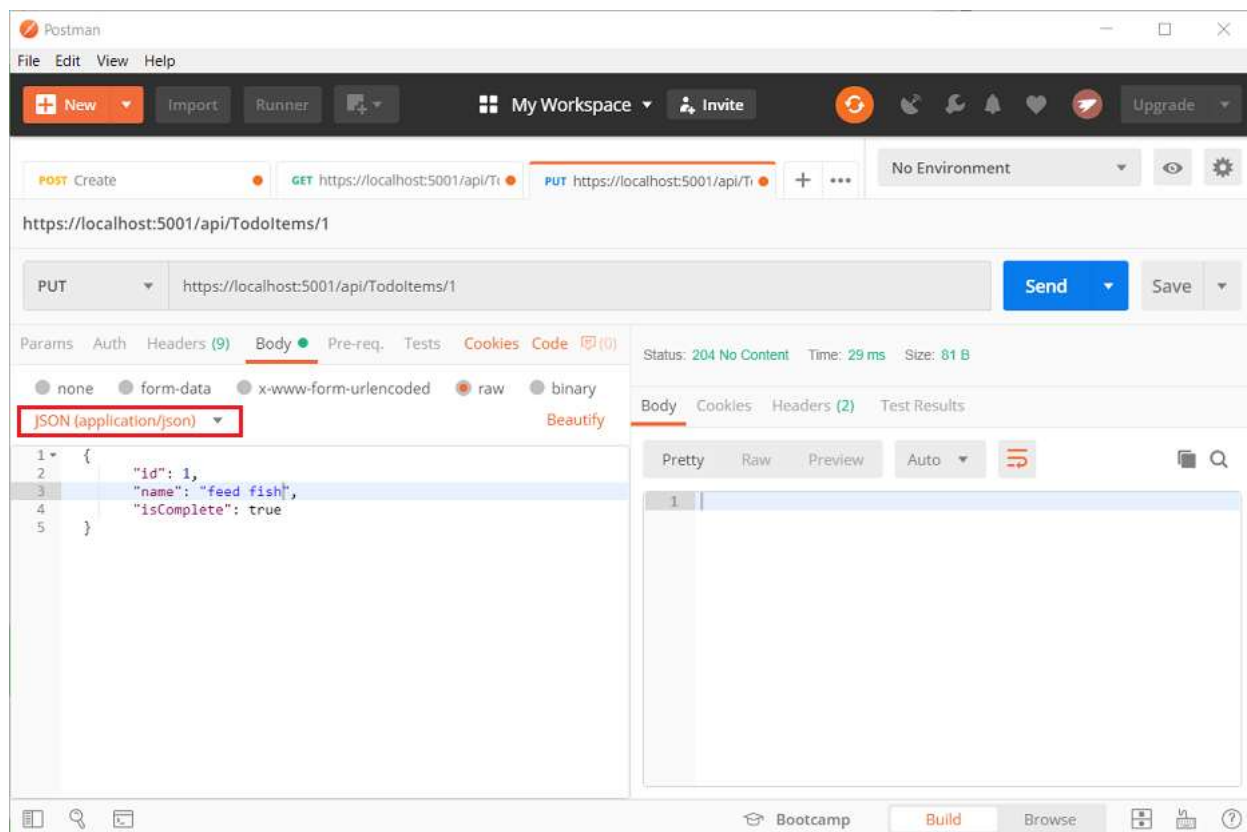
Тестирование метода `PutTodoItem`

В этом примере используется база данных в памяти, которая должна быть инициализирована при каждом запуске приложения. При выполнении вызова PUT в базе данных уже должен существовать какой-либо элемент. Для этого перед вызовом PUT выполните вызов GET, чтобы убедиться в наличии такого элемента в базе данных.

Обновите элемент списка дел с идентификатором 1 и присвойте ему имя "feed fish":

JSON	 Копировать
<pre>{ "ID":1, "name":"feed fish", "isComplete":true }</pre>	

На следующем рисунке показан процесс обновления Postman:



Метод DeleteTodoItem

Проверьте метод DeleteTodoItem.

C#	Копировать
<pre>// 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; }</pre>	

Тестирование метода DeleteTodoItem

Удалите элемент списка дел с помощью Postman:

- Укажите метод DELETE.

- Укажите URI удаляемого объекта (например, `https://localhost:5001/api/ToDoItems/1`).
- Нажмите кнопку **Отправить**.


Предотвращение избыточной публикации

В настоящее время пример приложения предоставляет весь объект `ToDoItem`. Рабочие приложения обычно ограничивают вводимые данные и возвращают их с помощью подмножества модели. Это связано с несколькими причинами, и безопасность является основной. Подмножество модели обычно называется объектом передачи данных (DTO), моделью ввода или моделью представления. В этой статье используется **DTO**.

DTO можно использовать для следующего:

- Предотвращение избыточной публикации.
- Скрытие свойств, которые не предназначены для просмотра клиентами.
- Пропуск некоторых свойств, чтобы уменьшить размер полезной нагрузки.
- Сведение графов объектов, содержащих вложенные объекты. Сведенные графы объектов могут быть удобнее для клиентов.

Чтобы продемонстрировать подход с применением DTO, обновите класс `ToDoItem`, включив в него поле секрета:

C#	 Копировать
<pre>public class ToDoItem { public long Id { get; set; } public string Name { get; set; } public bool IsComplete { get; set; } public string Secret { get; set; } }</pre>	

Поле секрета должно быть скрыто в этом приложении, однако административное приложение может отобразить его.

Убедитесь, что вы можете отправить и получить секретное поле.


Создайте модель DTO:

C#	 Копировать
<pre>public class ToDoItemDTO { public long Id { get; set; }</pre>	

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

Обновите `TodoItemsController` для использования `TodoItemDTO`:

C#

 Копировать

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

[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);
}

[HttpPut("{id}")]
public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO
todoItemDTO)
{
    if (id != todoItemDTO.Id)
    {
        return BadRequest();
    }

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

    todoItem.Name = todoItemDTO.Name;
    todoItem.IsComplete = todoItemDTO.IsComplete;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
    {

```

```

        return NotFound();
    }

    return NoContent();
}

[HttpPost]
public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO
todoItemDTO)
{
    var todoItem = new TodoItem
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

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

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

[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) =>
    _context.TODOItems.Any(e => e.Id == id);

private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
    new TodoItemDTO
    {
        Id = todoItem.Id,
        Name = todoItem.Name,
        IsComplete = todoItem.IsComplete
    };
}

```

Убедитесь, что вы можете отправить или получить секретное поле.

Вызов веб-API с помощью JavaScript

См. руководство по [: Вызовите веб-API ASP.NET Core с помощью JavaScript.](#)

Добавление поддержки аутентификации в веб-API

Удостоверение ASP.NET Core позволяет использовать функцию входа в пользовательском интерфейсе для веб-приложений ASP.NET Core. Чтобы защитить веб-API и одностраничные приложения, используйте один из следующих способов:

- [Azure Active Directory](#)
- [Azure Active Directory B2C](#) (Active Directory B2C)]
- [IdentityServer4](#)

IdentityServer4 — это платформа OpenID Connect и OAuth 2.0 для ASP.NET Core 3.0. IdentityServer4 включает следующие функции безопасности:

- Проверка подлинности как услуга (AaaS)
- Единый вход (SSO) для нескольких типов приложений
- Контроль доступа для API
- Шлюз федерации

Дополнительные сведения см. в разделе [Добро пожаловать в IdentityServer4.](#)

Дополнительные ресурсы

[Просмотреть или скачать пример кода для этого учебника.](#) См. раздел [Практическое руководство. Скачивание файла.](#)

Дополнительные сведения см. в следующих ресурсах:

- [Создание веб-API с помощью ASP.NET Core](#)
- [Страницы справки по веб-API ASP.NET Core с использованием Swagger \(OpenAPI\)](#)
- [Razor Pages с Entity Framework Core в ASP.NET Core: учебник 1 из 8](#)
- [Маршрутизация к действиям контроллера в ASP.NET Core](#)
- [Типы возвращаемых значений действий контроллера в веб-API ASP.NET Core](#)
- [Развертывание приложений ASP.NET Core в Службе приложений Azure](#)
- [Размещение и развертывание ASP.NET Core](#)
- [Версия руководства на YouTube](#)

Были ли сведения на этой странице полезными?

 Да  Нет
