**Minimal API with ASP.NET Core**

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 creates the following API:

| **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 |
| DELETE /todoitems/{id} | Delete an item | None | None |

**Create an API project**

The Program.cs file contains the following code:

|  |
| --- |
| var builder = WebApplication.CreateBuilder(args);  var app = builder.Build();  app.MapGet("/", () => "Hello World!");  app.Run(); |

The preceding code:

* Creates a [WebApplicationBuilder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder) and a [WebApplication](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication) with preconfigured defaults.
* Creates an HTTP GET endpoint / that returns Hello World!:

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

* From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.
* Select the **Browse** tab.
* 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:

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

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

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  class TodoDb : DbContext  {  public TodoDb(DbContextOptions<TodoDb> options)  : base(options) { }  public DbSet<Todo> Todos => Set<Todo>();  } |

**Add the API code**

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

|  |
| --- |
| 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(); |

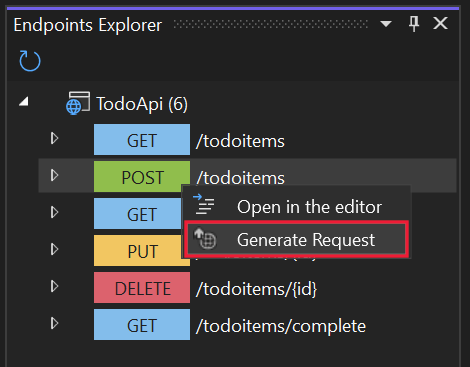
**Test posting data**

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

|  |
| --- |
| app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>  {  db.Todos.Add(todo);  await db.SaveChangesAsync();  return Results.Created($"/todoitems/{todo.Id}", todo);  }); |

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

* Select **View** > **Other Windows** > **Endpoints Explorer**.
* Right-click the **POST** endpoint and select **Generate request**.



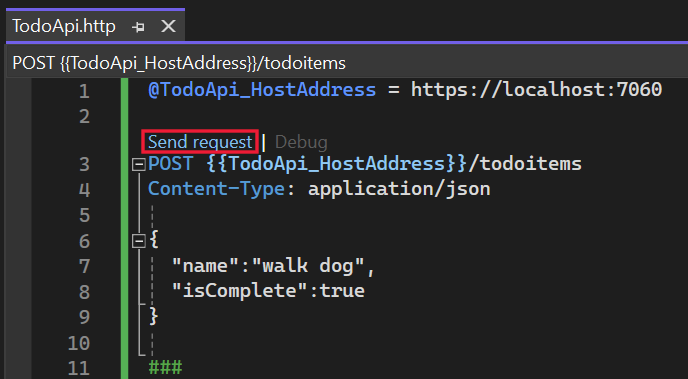
A new file is created in the project folder named TodoApi.http, with contents similar to the following example:

|  |
| --- |
| @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  } |

* 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.

A screenshot of a computer program

Description automatically generated

**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](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.endpointroutebuilderextensions.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](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.authorizationendpointconventionbuilderextensions.requireauthorization) and [WithMetadata](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.routingendpointconventionbuilderextensions.withmetadata).

**Use the TypedResults API**

Returning [TypedResults](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.typedresults) rather than [Results](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.results) has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the endpoint.

|  |
| --- |
| 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();  } |

**Choose between controller-based APIs and minimal APIs**

ASP.NET Core supports two approaches to creating APIs: a controller-based approach and minimal APIs. *Controllers* in an API project are classes that derive from [ControllerBase](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase). *Minimal APIs* define endpoints with logical handlers in lambdas or methods. This article points out differences between the two approaches.

The design of minimal APIs hides the host class by default and focuses on configuration and extensibility via extension methods that take functions as lambda expressions. Controllers are classes that can take dependencies via constructor injection or property injection, and generally follow object-oriented patterns. Minimal APIs support dependency injection through other approaches such as accessing the service provider.

|  |
| --- |
| namespace APIWithControllers;  public class Program  {  public static void Main(string[] args)  {  var builder = WebApplication.CreateBuilder(args);  builder.Services.AddControllers();  var app = builder.Build();  app.UseHttpsRedirection();  app.MapControllers();  app.Run();  }  } |

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace APIWithControllers.Controllers;  [ApiController]  [Route("[controller]")]  public class WeatherForecastController : ControllerBase  {  private static readonly string[] Summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };  private readonly ILogger<WeatherForecastController> \_logger;  public WeatherForecastController(ILogger<WeatherForecastController> logger)  {  \_logger = logger;  }  [HttpGet(Name = "GetWeatherForecast")]  public IEnumerable<WeatherForecast> Get()  {  return Enumerable.Range(1, 5).Select(index => new WeatherForecast  {  Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  TemperatureC = Random.Shared.Next(-20, 55),  Summary = Summaries[Random.Shared.Next(Summaries.Length)]  })  .ToArray();  }  } |

The following code provides the same functionality in a minimal API project. Notice that the minimal API approach involves including the related code in lambda expressions.

|  |
| --- |
| namespace MinimalAPI;  public class Program  {  public static void Main(string[] args)  {  var builder = WebApplication.CreateBuilder(args);  var app = builder.Build();  app.UseHttpsRedirection();  var summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };  app.MapGet("/weatherforecast", (HttpContext httpContext) =>  {  var forecast = Enumerable.Range(1, 5).Select(index =>  new WeatherForecast  {  Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  TemperatureC = Random.Shared.Next(-20, 55),  Summary = summaries[Random.Shared.Next(summaries.Length)]  })  .ToArray();  return forecast;  });  app.Run();  }  } |

Minimal APIs have many of the same capabilities as controller-based APIs. They support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. There are a few capabilities available with controller-based APIs that are not yet supported or implemented by minimal APIs. These include:

* No built-in support for model binding ([IModelBinderProvider](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.imodelbinderprovider), [IModelBinder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.imodelbinder)). Support can be added with a custom binding shim.
* No built-in support for validation ([IModelValidator](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.validation.imodelvalidator)).
* No support for [application parts](https://learn.microsoft.com/en-us/aspnet/core/mvc/advanced/app-parts?view=aspnetcore-8.0) or the [application model](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/application-model?view=aspnetcore-8.0). There's no way to apply or build your own conventions.
* No built-in view rendering support. We recommend using [Razor Pages](https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-8.0) for rendering views.
* No support for [JsonPatch](https://www.nuget.org/packages/Microsoft.AspNetCore.JsonPatch/)
* No support for [OData](https://www.nuget.org/packages/Microsoft.AspNetCore.OData/)