Build a Simple CRUD App with Angular 8 and ASP.NET Core 2.2 - part 1 - back-end

Martin Soderlund Ek Sep 18 · 6 min read

#dotnet #angular

Part 1 - back-end with ASP.NET Core 2.2 and Entity Framework Core

Let's take a look at how to build a CRUD web app using ASP.NET Core 2.2, Entity Framework Core, and Angular 8. This is part 1 where we focus on the back-end.

Part 2 is here - Angular 8 app with REST API

We'll have a REST API in the back-end and the Angular app on the front-end.

The basic CRUD app will be a blog, where we can **C**reate, **R**ead, **U**pdate, and **D**elete blog posts.

Github repo is here: Angular 8 blog app tutorial using .NET Core 2.2 and Entity Framework back-end

Prerequisites

- .NET Core 2.2 SDK
- Visual Studio 2019

For the Angular front-end we'll also use:

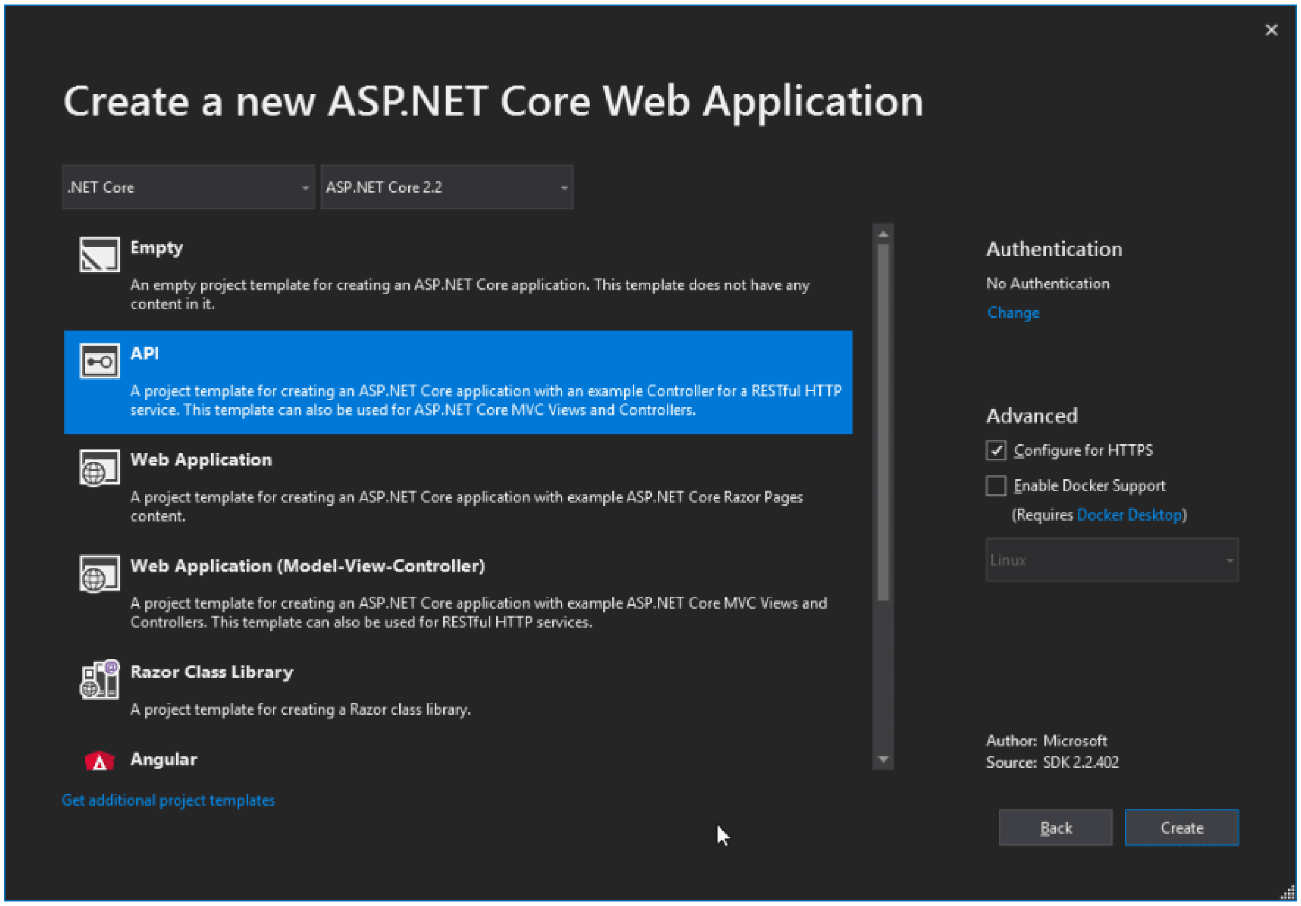
- VS Code
- Node.js
- Angular CLI

Make sure above are installed. We will use Visual Studio 2019 for the back-end and VS Code for the front-end. You can however use only Visual Studio 2019 if you want to.

As of Sept 15th, there are previews of .NET Core 3 that work with previews of Visual Studio 2019. However, in this tutorial we won't use any previews and only use fully released versions of .NET Core 2 and Visual Studio 2019.

Create ASP.NET Core 2.2 REST API

In Visual Studio 2019, create a new project and choose ASP.NET Core Web Application. Name our project (Blog). Then choose .ASP.NET Core 2.2 version and the API template:


Create a new ASP.NET Core Web Application

.NET Core | ASP.NET Core 2.2

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Razor Class Library
A project template for creating a Razor class library.

Angular

Get additional project templates

Authentication
No Authentication
Change

Advanced
[✓] Configure for HTTPS
[ ] Enable Docker Support
(Requires Docker Desktop)
Linux

Author: Microsoft
Source: SDK 2.2.402

Add models and Entity Framework Database Context

Next up, let's create a folder called Models, and then add a class file called **BlogPost.cs**. Make sure to import necessary namespaces.

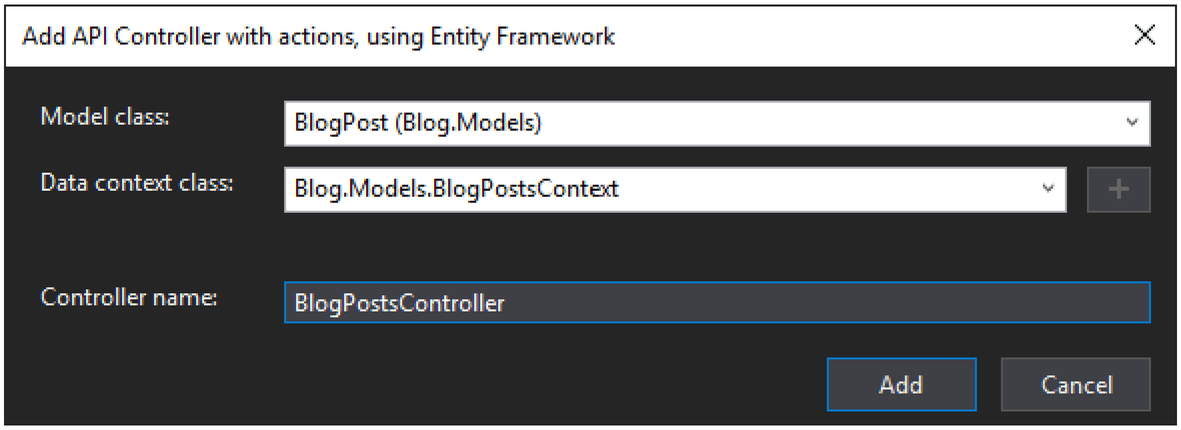
```
public class BlogPost
{
  [Key]
  public int PostId { get; set; }

  [Required]
```

```
    public string Creator { get; set; }

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

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

    [Required]
    public DateTime Dt { get; set; }
  }
```

BlogPost.cs hosted with ♡ by GitHub view raw

Then create an API controller — right click the Controllers folder, choose Add -> Controller. Then choose **API controller with actions, using Entity Framework**.

- Choose BlogPost as Model class.
- Under Data context class, press the + button and call the context BlogPostsContext.


Add API Controller with actions, using Entity Framework

Model class: BlogPost (Blog.Models)

Data context class: Blog.Models.BlogPostsContext

Controller name: BlogPostsController

When you press Add, Visual Studio will add the necessary NuGet packages and create a new database context, BlogPostContext, in the Data folder. You will also have the BlogPostsController created for you, filled with lots of API methods.

The contents of **BlogPostsContext.cs**:

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

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

BlogPostsContext.cs hosted with ♡ by GitHub view raw

The contents of newly created **BlogPostsController.cs**, our ApiController:

```
[Route("api/[controller]")]
[Produces("application/json")]
[ApiController]
public class BlogPostsController : ControllerBase
{
    private readonly BlogPostsContext _context;

    public BlogPostsController(BlogPostsContext context)
    {
        _context = context;
    }

    // GET: api/BlogPosts
    [HttpGet]
    public async Task<ActionResult<IEnumerable<BlogPost>>> GetBlogPost()
    {
        return await _context.BlogPosts.ToListAsync();
    }

    // GET: api/BlogPosts/5
    [HttpGet("{id}")]
    public async Task<ActionResult<BlogPost>> GetBlogPost(int id)
    {
        var blogPost = await _context.BlogPosts.FindAsync(id);

```

```
            if (blogPost == null)
            {
                return NotFound();
            }

            return blogPost;
        }

        // PUT: api/BlogPosts/5
        [HttpPut("{id}")]
        public async Task<IActionResult> PutBlogPost(int id, BlogPost blogPost)
        {
            if (id != blogPost.PostId)
            {
                return BadRequest();
            }

            _context.Entry(blogPost).State = EntityState.Modified;

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

            return NoContent();
        }

        // POST: api/BlogPosts
        [HttpPost]
        public async Task<ActionResult<BlogPost>> PostBlogPost(BlogPost blogPost)
        {
            _context.BlogPosts.Add(blogPost);
            await _context.SaveChangesAsync();

            return CreatedAtAction("GetBlogPost", new { id = blogPost.PostId }, blogPost)
        }

        // DELETE: api/BlogPosts/5
        [HttpDelete("{id}")]
        public async Task<ActionResult<BlogPost>> DeleteBlogPost(int id)
        {
            var blogPost = await _context.BlogPosts.FindAsync(id);
            if (blogPost == null)
            {
                return NotFound();
            }

            _context.BlogPosts.Remove(blogPost);
            await _context.SaveChangesAsync();

            return blogPost;
        }

        private bool BlogPostExists(int id)
        {
            return _context.BlogPosts.Any(e => e.PostId == id);
        }
    }
```

BlogPostsController.cs hosted with ♡ by GitHub view raw

This is fully working code, with route configuration, all CRUD operations and correct HTTP verbs using annotations (HttpPost, HttpGet, HttpPut, HttpDelete). We also force our API to serve JSON, using the `[Produces("application/json")]` filter.

We're following the best practices for REST APIs, as we use GET for listing data, POST for adding new data, PUT for updating existing data, and DELETE for deleting data.

Note that the BlogPostContext is **dependency injected**. This context is used to perform all actions needed for our app's back-end.

We can however improve this a little bit, using the Repository design pattern to create a data repository, and inject it.

Create a data repository

Our existing code works, however as applications grow, it's better to split the logic into different layers:

- Data layer with the data repository that communicates with the database.
- Service layer with services used to process logic and data layer communication.
- Presentation layer with only the API controller.

For our application, we will have an API controller which communicates with the data repository. Let's create the repository.

Right click the Data folder and create a new interface called IDataRepository. Copy and paste this code into IDataRepository.cs:

```
public interface IDataRepository<T> where T : class
{
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
    Task<T> SaveAsync(T entity);
}
```

IDataRepository.cs hosted with ♡ by GitHub view raw

Then right click the Data folder again and create a new class called DataRepository. Copy and paste this code into **DataRepository.cs**:

```
public class DataRepository<T> : IDataRepository<T> where T : class
{
    private readonly BlogPostsContext _context;

    public DataRepository(BlogPostsContext context)
    {
        _context = context;
    }

    public void Add(T entity)
    {
        _context.Set<T>().Add(entity);
    }

    public void Update(T entity)
    {
        _context.Set<T>().Update(entity);
    }

    public void Delete(T entity)
    {
        _context.Set<T>().Remove(entity);
    }

    public async Task<T> SaveAsync(T entity)
    {
        await _context.SaveChangesAsync();
        return entity;
    }
}
```

DataRepository.cs hosted with ♡ by GitHub view raw

As you can see, we have dependency injected the BlogPostContext into our DataRepository class.

Update BlogPostsController to use the data repository

Replace the code in BlogPostsController with the following:

```
[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class BlogPostsController : ControllerBase
{
    private readonly BlogPostsContext _context;
    private readonly IDataRepository<BlogPost> _repo;

    public BlogPostsController(BlogPostsContext context, IDataRepository<BlogPost> r
    {
        _context = context;
        _repo = repo;
    }

    // GET: api/BlogPosts
    [HttpGet]
    public IEnumerable<BlogPost> GetBlogPosts()
    {
        return _context.BlogPosts.OrderByDescending(p => p.PostId);
    }

    // GET: api/BlogPosts/5
    [HttpGet("{id}")]
    public async Task<IActionResult> GetBlogPost([FromRoute] int id)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var blogPost = await _context.BlogPosts.FindAsync(id);

```

```
            if (blogPost == null)
            {
                return NotFound();
            }

            return Ok(blogPost);
        }

        // PUT: api/BlogPosts/5
        [HttpPut("{id}")]
        public async Task<IActionResult> PutBlogPost([FromRoute] int id, [FromBody] Blog
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            if (id != blogPost.PostId)
            {
                return BadRequest();
            }

            _context.Entry(blogPost).State = EntityState.Modified;

            try
            {
                _repo.Update(blogPost);
                var save = await _repo.SaveAsync(blogPost);
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!BlogPostExists(id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }

            return NoContent();
        }

        // POST: api/BlogPosts
        [HttpPost]
        public async Task<IActionResult> PostBlogPost([FromBody] BlogPost blogPost)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            _repo.Add(blogPost);
            var save = await _repo.SaveAsync(blogPost);

            return CreatedAtAction("GetBlogPost", new { id = blogPost.PostId }, blogPost
        }

        // DELETE: api/BlogPosts/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteBlogPost([FromRoute] int id)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

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

            _repo.Delete(blogPost);
            var save = await _repo.SaveAsync(blogPost);

            return Ok(blogPost);
        }

        private bool BlogPostExists(int id)
        {
            return _context.BlogPosts.Any(e => e.PostId == id);
        }
    }
```

BlogPostsController.cs hosted with ♡ by GitHub view raw

You can see the data repository in action in the `PutBlogPost`, `PostBlogPost` and `DeleteBlogPost` methods, like this:

```
_repo.Add(blogPost);
var save = await _repo.SaveAsync(blogPost);
```

BlogPostsController.cs sample hosted with ♡ by GitHub view raw

We however choose to keep our dependency on BlogPostsContext in the controller, using both the context and data repository.

CORS in app configuration

In **Startup.cs**, you already can see some configuration — our app will use MVC and have a db context for instance. Update the ConfigureServices method to look like this:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

  services.AddDbContext<BlogPostsContext>(options =>
          options.UseSqlServer(Configuration.GetConnectionString("BlogPostsContext"))

  services.AddCors(options =>
  {
      options.AddPolicy("CorsPolicy",
          builder => builder.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader()
              .AllowCredentials());
  });

  services.AddScoped(typeof(IDataRepository<>), typeof(DataRepository<>));

  // In production, the Angular files will be served from this directory
  //services.AddSpaStaticFiles(configuration =>
  //{
  //    configuration.RootPath = "ClientApp/dist";
  //});
}
```

Startup.cs hosted with ♡ by GitHub view raw

Since we’re going to call our REST API from JavaScript, we need to enable CORS. We will use a default policy.

Also, we register our DataRepository here.

Make sure to import necessary namespaces.

Note that we’ve commented out `services.AddSpaStaticFiles()`. We will uncomment this when we’ve created the Angular application, but we’re not there yet.

Then update the Configure method to look like this:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    // The default HSTS value is 30 days. You may want to change this for production
    app.UseHsts();
  }

  app.UseCors("CorsPolicy");
  app.UseHttpsRedirection();
  app.UseStaticFiles();
  //app.UseSpaStaticFiles();

```

```
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller}/{action=Index}/{id?}");
    });

    //app.UseSpa(spa =>
    //{
    //    // To learn more about options for serving an Angular SPA from ASP.NET Core,
    //    // see https://go.microsoft.com/fwlink/?linkid=864501

    //    spa.Options.SourcePath = "ClientApp";

    //    if (env.IsDevelopment())
    //    {
    //        spa.UseAngularCliServer(npmScript: "start");
    //    }
    //});
}
```

Startup.cs hosted with ♡ by GitHub view raw

`UseCors` must come before `UseMvc` here. Don't forget to import the necessary namespaces.

We've commented out SPA specific configuration here. We will uncomment `app.UseSpaStaticFiles()` and `app.UseSpa()` later, when we've created our Angular application.

Let's also update **launchSettings.json** to set `launchUrl` to empty in two places:

`"launchUrl":` "”

Also delete **ValuesController.cs** in the Controllers folder.

Setup migrations and create the database

We're almost there!

Now that we have the BlogPostsContext and use the code first approach for Entity Framework, it's time to setup migrations. First, let's take a look at the database connection string in **appSettings.json**.

The connection string was created for us earlier and the app will use SQL Server Express LocalDb. You can, of course, use your own instance of SQL Server instead, just make sure the connection string is correct!

```
"ConnectionStrings": {
  "BlogPostsContext": "Server=(localdb)\\mssqllocaldb;Database=BlogPostContext-d08fc30
}
```

appSettings.json hosted with ♡ by GitHub view raw

To enable migrations, open the Package Manager Console (Tools->NuGet Package Manager->Package Manager Console) and run this command:

```
Add-Migration Initial
```

We'll get this message back:

```
Microsoft.EntityFrameworkCore.Infrastructure[10403]
```

```
Entity Framework Core 2.2.6-servicing-10079 initialized 'BlogPostsContext' usi

To undo this action, use Remove-Migration.
```

Perfect! A Migrations folder is created with your Entity Framework migrations, which will contain changes to the Entity Framework model. We have one migration file, named something like **20190912150055_Initial.cs**

In this file, we have the `Up` and `Down` methods, which will upgrade and downgrade the database to the next or previous version.

To create the database, we now have to execute the following command in the Package Manager Console:

```
Update-Database
```

Let's open the SQL Server Object Explorer (View -> SQL Server Object Explorer). We now have this:

Remember, if you make any changes to the data model, you need to use the `Add-Migration YourMigrationName` and `Update-Database` commands to push changes to the database.

Try the API

Press F5 in Visual Studio to start the application. On localhost, browse to `/api/blogposts` which should return an empty JSON string. You can now use Postman to create a new blog post.

Here’s the JSON to post:

```
{
  "dt": "2019-09-12T18:18:02.190Z",
  "creator": "Martin",
  "title": "Test",
  "body": "Testing"
}
```

In Postman, make sure the API URL is correct and POST is used as the http verb. Fill in above JSON in the editor (choose raw) and choose JSON (application/json) before you press Send. The request and returned body result should look like this:

And if you change http verb from POST to GET, you’ll now get this result:

Our API is up and running!

Now on to our Angular 8 front-end app. Here's part 2 of the tutorial. Part 2 is here - Angular 8 app with REST API

Are you new to dev.to?

Making the most of dev.to

Thanks for stopping by ♡

Martin Soderlund Ek **+ FOLLOW**

Full stack .NET developer, cyclist, father. Curious and happy!

@dileno dileno dileno

Add to the discussion

PREVIEW SUBMIT

code of conduct - report abuse

Classic DEV Post from Jun 30

What is your favourite Git command?

Muna Mohamed

94 57

Another Post You Might Like

"Differential Loading" - A New Feature of Angular CLI v8

Suguru Inatomi

46 12

Another Post You Might Like

Building scalable robusts and type safe forms with Angular

Maxime

159 17

Continuous deployment to Azure from Azure DevOps

Thomas Ardal - Nov 18

Top 5 Front-End JavaScript Frameworks & Libraries in 2019

SyedSami-UI - Nov 18

Blazor-Auth0 v2.0.0-Preview4 is alive!

Henry Alberto Rodriguez - Nov 17

No more 💩 angular unit test / #2 Component logic

Alexandre - Nov 17

Home About Privacy Policy Terms of Use Contact Code of Conduct

DEV Community copyright 2016 - 2019 🔥