

# APIs ReST

- [Controlleurs et Endpoints](#)
- [Tests d'intégration](#)
- [Swagger UI](#)

# Controlleurs et Endpoints

Controllers are the classes that handle HTTP Requests by holding method that correspond to API endpoints. First, create a *Controllers* folder to store you controllers. To create a controller, create a class with the following naming convention : the name of the resource, starting with a uppercase letter, then *Controller*.

Then, the class must extend the `ControllerBase` class and be annotated with the `[ApiController]` and `[Route]` annotation :

- `[ApiController]` enables automatic model validation, we'll come to that later
- `[Route]` maps the controllers Route, takes a string parameter to map the route

Example :

```
namespace TodoAPI.Controllers {  
    [ApiController]  
    [Route("api/[controller]")]  
    public class TodosController : ControllerBase {  
    }  
  
}
```

This controller handles request that start with : *https://mytodo.io/api/Todos*

To handle requests, a controller must have API endpoints method. Those method handle HTTP requests and return HTTP responses.

To be an enpoint method, a method must be annotated with a HTTP Verb annotation have a IActionResult return type. The HTTP Verb annotation take an optionanl parameter for the route mapping. If this argument is not provided, the method with handle the HTTP verb for the Controller's route.

Let's say we have a model object like that in our Models folder :

```
namespace TodoAPI.Models {  
    public class Todo {  
        public string Name {get;set;}  
        public string Description {get;set;}  
    }  
}
```

```
}
```

## Verb Mapping

You can just return the response object Within the `Ok()` method that will take care of returning the right HTTP status code and serialize the object to JSON.

Example :

```
[HttpGet]
public IActionResult GetTodos() => Ok( new List<Todo>(){new Todo(){Name = "Hello",
Description = "World"}});
```

This method will handle GET requests on the endpoint */api/Todos*

## Body Parameters

You can get a parameter from the request body :

```
[HttpPost]
public IActionResult CreateTodo([FromBody] Todo todo){

}
}
```

## Query Parameter

This method will handle POST requests on the endpoint */api/Todos* and have a Todo JSON representation as body.

You can also retrieve a GET query parameter :

```
[HttpGet]
public IActionResult SearchTodos([FromQuery("name")] string name){

}
}
```

## Path Parameters

You can also retrieve a path parameter :

```
[HttpGet("{todoId:Guid}")]  
public IActionResult GetTodo(Guid todoId){  
  
}
```

# Tests d'intégration

Integration testing are usefull for validating user stories automatically. It is usefull for this test which user all the layers of the app including the database connection, to use a InMemory database.

## Create Custom WebApplicationFactory

```
public class CustomWebApplicationFactory : WebApplicationFactory<Startup>
{
    public ApplicationDbContext Context { get; private set; }

    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Remove or app real DbProvider from the DI container
            var descriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));
            services.Remove(descriptor);

            // Create a new service provider for the InMemory Database
            var serviceProvider = new ServiceCollection()
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Add a database context (AppDbContext) using an in-memory database for
testing.

            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryAppDb");
                options.UseInternalServiceProvider(serviceProvider);
            });
        });
    }
}
```

```

        // Build the service provider
        var sp = services.BuildServiceProvider();

        // Create a scope to obtain a reference to the database contexts (for
verification in testing)
        var scope = sp.CreateScope();
        var scopedServices = scope.ServiceProvider;
        var appDb = scopedServices.GetRequiredService<ApplicationDbContext>();

        var logger =
scopedServices.GetRequiredService<ILogger<CustomWebApplicationFactory>>();

        // Ensure the database is created (ensure migrations are executed)
        appDb.Database.EnsureCreated();
        Context = appDb;
    });
}
}

```

## Use the new **WebApplicationFactory** in a test

```

public partial class IntegrationTests : IClassFixture<CustomWebApplicationFactory>
{
    private readonly HttpClient client;
    private readonly ApplicationDbContext db;

    public IntegrationTests(CustomWebApplicationFactory factory) // Constructor is
executed between each test method
    {
        this.client = factory.CreateClient();
        this.db = factory.Context;
        factory.Context.Database.EnsureDeleted(); // Ensure the database is emptied
between each test method
    }
}

```

```

[Fact]
public async void Register()
{
    // Given
    var registerDto = new RegisterDT0()
    {
        Email = "john.shepard@n7.citadel",
        Username = "Shepard",
        Password = "NormandyTali<3"
    };

    // When
    var registerResponse = await client.PostAsJsonAsync("api/auth/register",
registerDto);

    // Then
    var result = await registerResponse.Content.ReadAsStringAsync();
    Assert.True(registerResponse.IsSuccessStatusCode);
    Assert.Equal(1,db.Set<User>().Count());
}
}

```

# Swagger UI

## Nugget Dependency

Install the Nugget package `Swashbuckle.AspNetCore` :

```
dotnet add Swashbuckle.AspNetCore
```

## Enable XML Documentation Generation

In the `.csproj` file of your project, add the following tags as child of the `<project>` tag to enable XML comments generation :

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

## Configuration

In `Startup.ConfigureServices` :

```
services.AddSwaggerGen(options =>
{
    /// API Metadata
    options.SwaggerDoc("ExampleAppDocumentation", new OpenApiInfo()
    {
        Title = "Example API",
        Description = "Backend for Example App",
        Version = "1",
        Contact = new OpenApiContact()
        {
            Email = "john.shepard@n7.al",
            Name = "John Shepard",
```



```

        Url = new Uri("https://arsenelapostolet.fr")
    },
    License = new OpenApiLicense()
    {
        Name = "MIT License",
        Url = new Uri("https://en.wikipedia.org/wiki/MIT_License")
    }
});
// Set the comments path for the Swagger JSON and UI.
var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
options.IncludeXmlComments(xmlPath);

```

□□□□// Add support for JWT Auth in SwaggerUI

```

options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
{
    Description = "JWT Bearer Authorization",
    Name = "Authorization",
    In = ParameterLocation.Header,
    Type = SecuritySchemeType.ApiKey,
    Scheme = "Bearer"
});
options.AddSecurityRequirement(new OpenApiSecurityRequirement()
{
    {
        new OpenApiSecurityScheme
        {
            Reference = new OpenApiReference
            {
                Type = ReferenceType.SecurityScheme,
                Id = "Bearer"
            },
            Scheme = "oauth2",
            Name = "Bearer",
            In = ParameterLocation.Header
        },
        new List<string>()
    }
});
});

```

In `Startup.Configure`, just after `app.UseHttpsRedirection();` :

```
app.UseSwagger();
app.UseSwaggerUI(options =>
{
    options.SwaggerEndpoint("/swagger/ExampleAppDocumentation/swagger.json",
"Example API");
    options.RoutePrefix = "";
});
```

That's it !