CI/CD API

Contents

How does it work?

Let's see step by step how does this API work.

Setup

First of all, we have to configure the following settings on appsettings.json file:

- ConnectionStrings.defaultConnection: The connection string to our database.
- AppSettings.workingDirectory: The path to our local folder in which this API is gonna clone repos and work with them.
- AppSettings.dockerizerFullFileName: The full path to the powershell script that creates images and deploys our projects to our Docker Container.
- AppSettings.useWsl: A boolean value indicating whether our Docker is working on Linux via WSL or not.

Second of all, we have to publish the database using the CICD.SQL project.

Configuration

On User Controller you see CRUD endpoints.
This Controller manages GitHub users on our local database.
"Create" endpoint asks for a Name and a Token. This token has to be generated on GitHub website (see https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token for more information).
Once the user name and its token are validated, a new record is created on [User] table.

On Project Controller you see CRUD endpoints.
Project is a GitHub repo.
"Create" endpoint asks for several parameters, including a *User* object for validation purposes, in order to configure how this repo is gonna be built, tested and deployed.

Parameters (properties of a *Project* object):

- User: A *User* object.
- Name: The name of the repo.
- RelativePath: The project file (i.e. .csproj file) relative path to the GitHub repo.
- DotnetVersion: The dotnet version used to build our project.
- Test: A boolean value indicating whether we want to run tests after building the project or not.
- Deploy: A boolean value indicating whether we want to deploy our project on a Docker Container or not.
- DeployPort: The port number where this project is gonna be available on Docker.

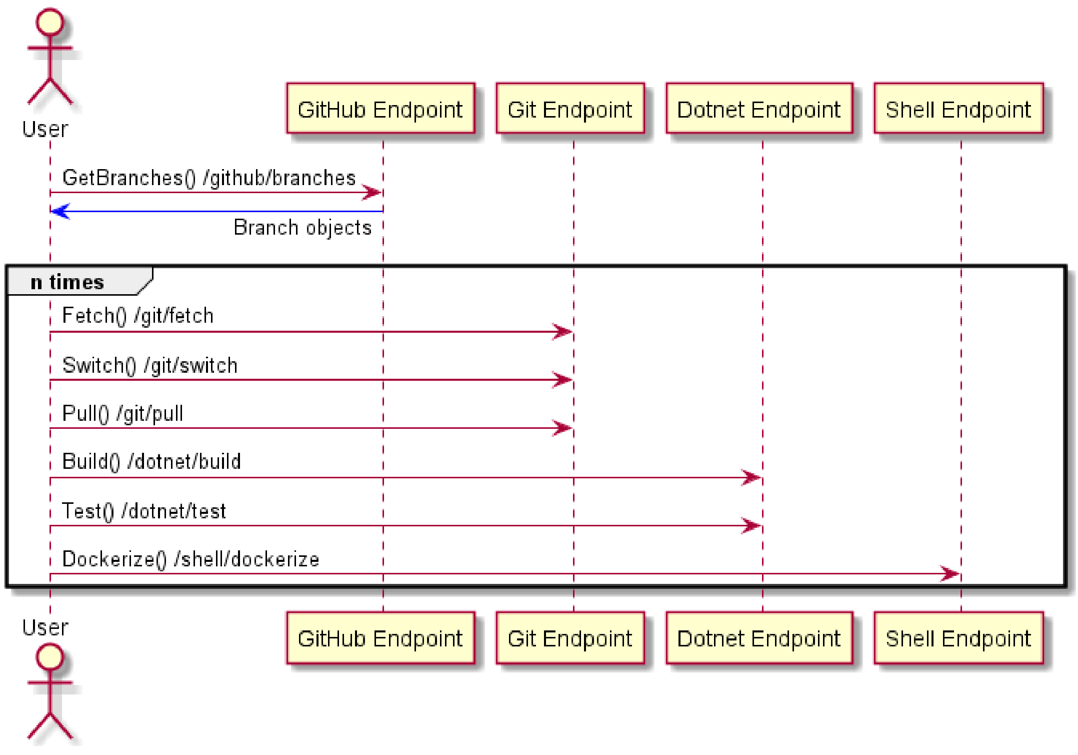
Once the project is validated, a new record is created on [Project] table.

For example, let's say we want to configure this .NET CI/CD API software to be cloned locally and then built it. The *Project* object to send to this endpoint is:

```
{
   user: {
      "name": "rlrecalde",
      "token": "******"
   },
   "name":  "CICD",
   "relativePath": "CICD/CICD.csproj",
   "dotnetVersion": "6.0",
   "test": false,
   "deploy": false,
   "deployPort": 0
}
```

Finally, given this *Project* object, we call Clone() method from Git Controller (/git/clone) in order to clone this repo on our local machine.

Running the CI/CD process



Given a *Project* object, we call GetBranches() method from GitHub Controller (/github/branches).
A collection of *Branch* objects are gonna be returned.
Each of these objects has a *Name* and a *LastCommit* property.
In case we know this last commit is equal to the last commit we got the last time we called to this endpoint, we don't have to do anything else because that means there are no changes on the repo. Also, we have to check if new *Branch* objects were returned since the last time we called to this endpoint.
In case new branches or new last commits were returned, we have to continue with next steps.

Optional: Given a *Branch* object, we may call GetLastCommitByBranch() method from GitHub Controller (/github/last-commit) in order to get more information about this commit (committer name, date, etc.), in case we want to show these data on a screen.

For each *Branch* object we want to work on:
We call Fetch() method from Git Controller (/git/fetch).
Then, the Switch() method from Git Controller (/git/switch).
Then, the Pull() method from Git Controller (/git/pull).
Then, the Build() method from Dotnet Controller (/dotnet/build).
Then, if we want to run tests, we call Test() method from Dotnet Controller (/dotnet/test).
Finally, if we want to deploy to Docker, we call Dockerize() method from Shell Controller (/shell/dockerize).

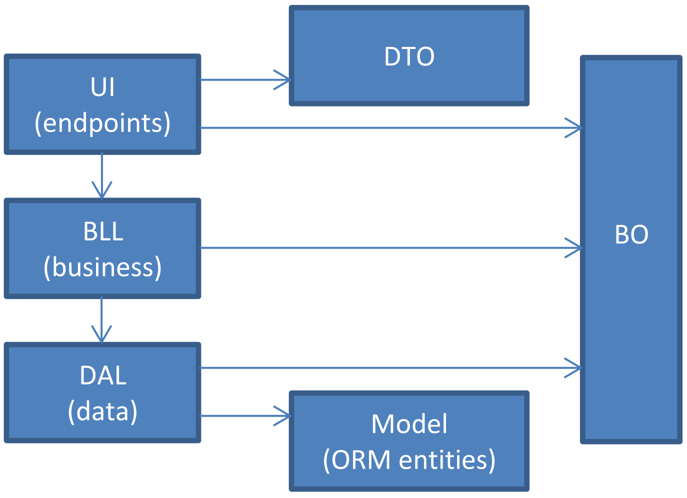
Also, at any time of this process, we can call Create() method or CreateMany() method from Branch Controller and Create() method from Commit Controller in order to persist our branches and commits information on the database. This is useful for getting them back next time we want to run the process and "know" what we did the last time we ran it.

My best practices

kebab-case

All endpoint names were implemented in kebab-case (i.e. /github/last-commit).

Layers & Visibility



Description

UI layer (Controllers and their endpoints) only sees BLL layer (Business Logic Layer).
BLL layer only sees DAL layer (Data Access Layer).
DTO (Data Transfer Objects), BO (Business Objects) and Model layers are just POCO entities.

DAL layer accesses the database with Model entities and communicates to BLL layer with BO entities.
BLL layer communicates to DAL and UI layers with BO entities (it is not the best practice to make BLL see or work with ORM objects).
UI Layer communicates to the outside world with DTO entities.

The reason I use BO entities is to be able to isolate our business logic from any kind of technology we use for accessing data.

Besides, it's important to notice that referenced layers (projects) on a .NET project don't have their own referenced layers (projects). That is, every layer only sees what it must see.
For instance, BLL layer sees DAL layer but does not see Model layer. We can check this out by expanding "Dependencies" tree on "CICD.BLL" project.
By Default, .NET Core permits this "cascade" manner of visibility. So, I was careful on configuring every .csproj file to avoid this.

Finally, "data access" doesn't mean "database access" only. That's why I also access external API's (i.e. GitHub API) and console commands (i.e. git console commands) from DAL layer.

Validations

We can see on some User and Project business methods I made some validations.
Validations were implemented on separate classes, in order to decouple these responsibilities from main business logic. It is not the best practice to code "if's" statements on the same business logic methods for validating purposes.

Async vs. not Async

We know asynchronous methods (with all of this "Task" and "await" stuff) enhance performance on some cases.
Many developers tend to make all of their endpoints async as a rule of thumb, nowadays.
We have to take into account that every time we define a method as a "Task", we are creating a new thread to execute it.
The more threads we have on our application, the less performance we get from it.
So, on this API, only the executions we need to run on a separate thread were implemented asynchronously (i.e. console commands execution on a System.Diagnostic.Process object).

Custom Exceptions

Validations, as well as certain responses from database, raise custom exceptions that are taken and processed by a Middleware in order to return specific HTTP codes as responses from requests to this API.
I consider, for instance, it is not the best practice to return a "null" object when we find nothing on the database, as a result of a certain query. Instead, I return a "Not Found" HTTP response (HTTP code = 404).
This same rule applies when we try to update or delete a record that does not exist on the database. That's why, for all of these cases, DAL methods raise a custom exception on these situations.

Middlewares

On the main project, inside "Filters" folder, we've got two middlewares: ApiActionFilter and ApiExceptionFilter.

ApiActionFilter middleware is used to log HTTP requests and responses. Usually, I use this filter to log to Open Tracing (i.e. Jaeger) on a micro-services architecture. In this case, I just log to the console.
It's important to make it log request payloads and response objects.

ApiExceptionFilter middleware is used to manage custom exceptions and has the responsibilities of returning different HTTP response codes and logging errors to the console.

Mappings

Given that we work with three different POCO entities (DTO, BO and Model), at some point we need to map/transform one object to another.

We know third parties mappers are very common and often used on the Industry, such as AutoMapper. But, let's think about this:
How many times we struggle with mapping configuration issues?
How much time we use to spend on fixing bugs regarding to these issues?
And what happens when we modify the property name or type of an object which is being mapped to another? (spoiler alert: our software builds fine, but does not work as expected).
Besides, we have to take into account that this kind of mappers work via Reflection, and this is not so optimal for our performance.
For all of these reasons, I just made my own mapping classes.
We can see them in "Mappers" folders on main project and DAL layer.

Logging

To make this API simple, I just log information (only for console command executions and HTTP requests and responses) and errors to the console.
As I mentioned before, errors are logged by ApiExceptionFilter middleware. But this is not the best practice. I did it like this just because I am logging everything to the console. The best practice is to make our business layer to accomplish this task, maybe making it derive from a base class that manages those custom exceptions and logs their errors.

But the more important thing we have to take into account is that this errors logging must include objects. Which objects? Those that were related to the error.
For instance, if we take a look at Update() method from Project business class, we can see I am raising an "UnexpectedException" on the second "catch" which includes the "project" object this method receives as parameter.
In this way, every time we see a logged error, we can have much more context on what happened; basically, we can reproduce the failing scenario using the same parameters that caused the error.

Dapper

As we know, Dapper is a micro-ORM. That means it is almost an ADO.NET tool to communicate to a database. Basically, it has the ability to map data results from queries into custom objects.
But what about the string sentence we have to send to the database (i.e. "select * from …")?

Given that we are object oriented and strongly typed developers, it seems Dapper does not fulfill our needs. But, on the other hand, its performance is way much better than EntityFramework and NHibernate.

That's why a "QueryBuilder" fluent class was developed and been used by DAL classes.
We can see it in action on every method of DAL layer that accesses the database.

Naming

Naming classes is important for understanding "what an object is".
Several years ago, when layers were not so popular, we've learned object oriented programming as the way of making an object to accomplish functionalities and to communicate with other objects through its properties and behavior.

Back in that days, for instance, an object of type "Person" used to have properties (name, age, address, etc.) and behavior (run, jump, eat, etc.).
Then, when we decided to separate concerns and responsibilities into layers (multitier architecture), we found out that we may have got many objects representing a "Person":

- A Person with only properties for communicating with the outside world.
- A Person with behavior for resolving business logic.
- A Person with another kind of behavior for persisting information on a data source.
- A Person with only properties that represents that data source schema.

Therefore, many developers tend to define different class names for each of those ("PersonRequest", "PersonResponse", "PersonDto", "PersonBusiness", "PersonData", etc.).
But all of these objects (I mean, the instances of those classes) still are <u>Persons</u>!
So, why do we change the name of what a thing is?
All of those classes are just Persons separated in layers. That's all.

That's why all of the classes used in this API has the same name but different namespace.
For instance, we have "DTO.User", "BLL.User", "BO.User", "Validators.User", "DAL.User" and "Model.User".
And then, let's say on BLL layer, the instance of a "DAL.User" class is called "userData", the instance of a "Validators.User" class is called "userValidator" and the instance of a "BO.User" class is called just "user".

More DTO's?

Finally, starting at C# 9 on .NET 5, we have "records" as immutable class types for, among other things, API calling purposes.
Until that, I used to define more DTO entity layers for communicating with other external API's (one for each API).
Now, as we can see on DAL.GitHub class, several records were defined to catch responses from the GitHub API.