

CI/CD Web UI

Contents

- How does it work? 2
 - Setup 2
 - First launch 2
 - Further launches 5
 - Error messages 5
- My best practices 6

How does it work?

This SPA is meant to work with CI/CD API.

We are going to assume that we've already read that API documentation.

It is not just a simple web application to show the CI/CD process. It also has the responsibility of making the process to work.

In the real world, we need to have a third application acting as a "process manager" in charge of making the process to work, in order to isolate this responsibility from just showing on a screen what's going on. But, just to make it simple for a demonstration, this SPA is in charge of doing all the job.

Setup

We've got just two settings to configure on config.json file:

- baseAPIUrl: The CI/CD API base URL. By default, both the API and this SPA are configured on "https://localhost:7142". So, we don't have to do anything to make them work together.
- gitHubPollingTimeSecs: Amount of time in seconds for polling the API for the process purpose.

First launch

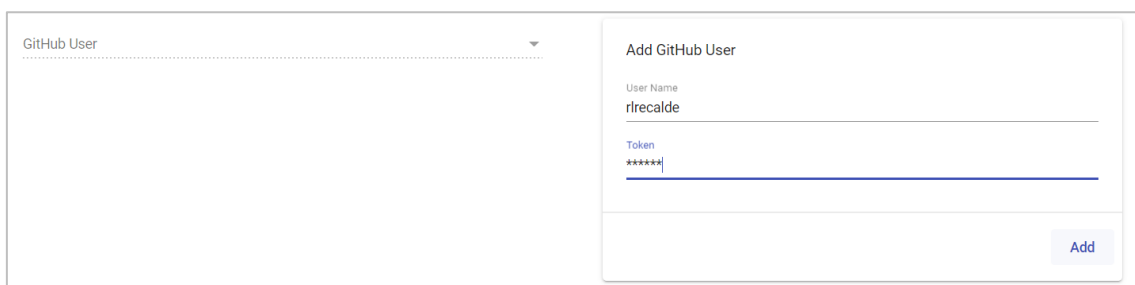
Once we launch this web application for the first time (and, of course, having our API running), we are presented to configure our GitHub User.



The screenshot shows a web interface with a horizontal container. On the left, there is a dropdown menu labeled 'GitHub User' with a downward arrow. To its right is a rectangular button labeled 'Add GitHub User'.

In fact, we are presented to select a User from the drop down list, but this list is empty for now.

So, clicking on the "Add GitHub User" panel we can configure our User.



The screenshot shows the 'Add GitHub User' form. It has a title 'Add GitHub User' at the top. Below it are two input fields: 'User Name' with the value 'rircalde' and 'Token' with the value '*****'. At the bottom right of the form is a blue button labeled 'Add'.

After that, our User is validated and added to our database. So now, we can select it from the drop down list.

The screenshot shows a user interface with a dropdown menu for 'GitHub User' set to 'rlrecalde'. To the right is a button labeled 'Add GitHub User'. Below the dropdown is a button labeled 'Set as Default'. Underneath is a section titled 'Projects' containing a button labeled 'Add Project (GitHub Repository)'.

If we don't want to select it every time we launch this application, we can click on "Set as Default" button in order to set it up as default (this will create a record on our [Configuration] database table).

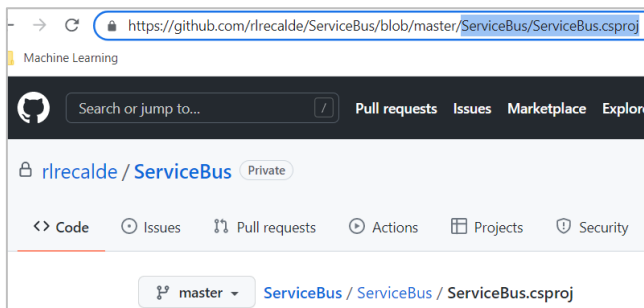
Now we are ready to add projects (GitHub repos). Just click on "Add Project (GitHub Repository)" panel.

This screenshot shows the 'Add Project (GitHub Repository)' panel expanded. It contains a form with the following fields: 'Name' (ServiceBus), 'Dotnet Version' (6.0), 'Relative Path' (ServiceBus/ServiceBus.csproj), and checkboxes for 'Test' and 'Deploy'. There is also a 'Deploy Port' field with the value 0. An 'Add' button is at the bottom right of the panel.

We may notice that every time we expand this panel, this application makes a request to the API. This is for getting dotnet versions installed on our local machine, in order to fulfil the drop down list located at the top right corner of this panel.

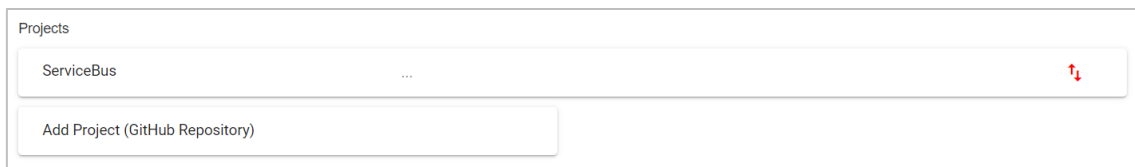
"Relative Path" may be confusing.

We can just navigate to our GitHub repository, look for our main .csproj file and copy and paste its relative path.

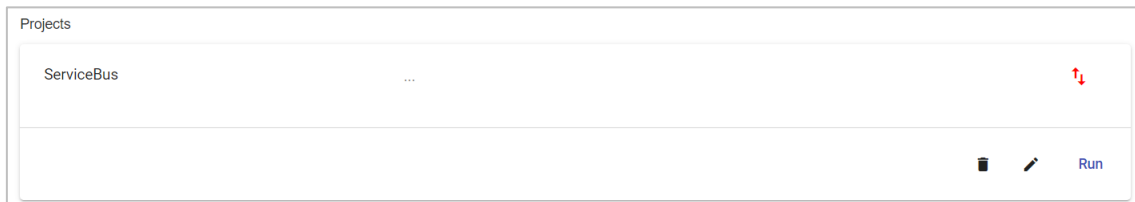


After clicking on "Add" button, this project will be validated and then this application will try to clone it to our local working directory.

If everything was fine, we will be presented to our new Project panel.

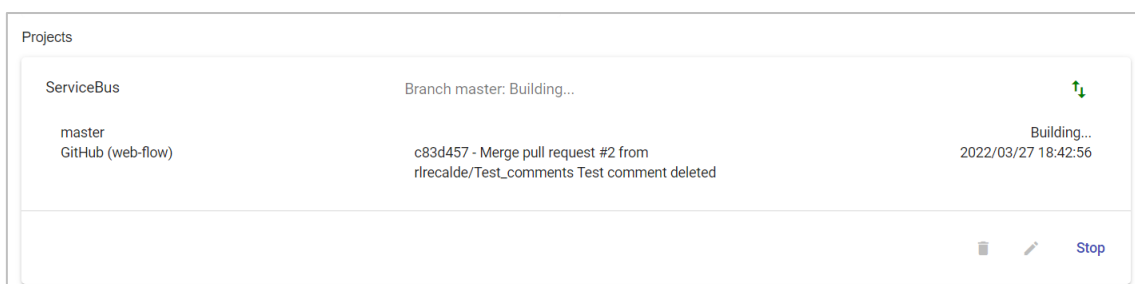


Every time we add a Project, a new panel is added.
So now, we are going to expand this panel and see what we have.



By clicking on "Run" button the process will start.
Also, we have a "delete" and "edit" button. For instance, if we realized that we have configured a wrong dotnet version and our project is not building correctly, we may want to "edit" it and fix our mistake.

So, let's run the process.



First, it gets branches from GitHub and shows them on the panel body.
Then, it saves these branches on the database.
Then, for each branch, it gets its last commit information, shows it on the panel body and saves it on the database.
Finally, it starts the process by fetching, switching, pulling and building each of the branches.
After the configured period of time elapsed, it re-runs the process all over again.

Given that we may be using a free version of GitHub account, we can't configure a branch as protected (with restrictions). So, we need to have some way to prevent merging branches before we are sure they build correctly.
We know every time we push commits to a branch, GitHub makes a Pull Request available to us. And, after a Pull Request is made, we are presented to the "Merge" button.
Therefore, in order to avoid clicking on that button before knowing if our branch builds correctly, after the building process finishes, this application sends a comment to its last commit saying "Build Ok!". This way, we may want to wait until this comment appears on GitHub Pull Request screen to do the merge.
Also, in case we configured to run tests on our project, another comment is sent saying "Test Ok!". This way, we may want to wait until both comments appear on the screen for merging the branch.

Finally, in case we have configured our project to be deployed on our local Docker Container, after building it (or running tests), it will check if this branch is the head branch. In case it is the head branch, it will make the deployment; otherwise, process ends.

Further launches

Once we have knowledge on our database regarding to branches and last commits related to a project, every time we launch this application and run the process, it will check the information returned by the API and compare it against this persisted information in order to decide whether it has to pull new branches and/or commits or not.

Error messages

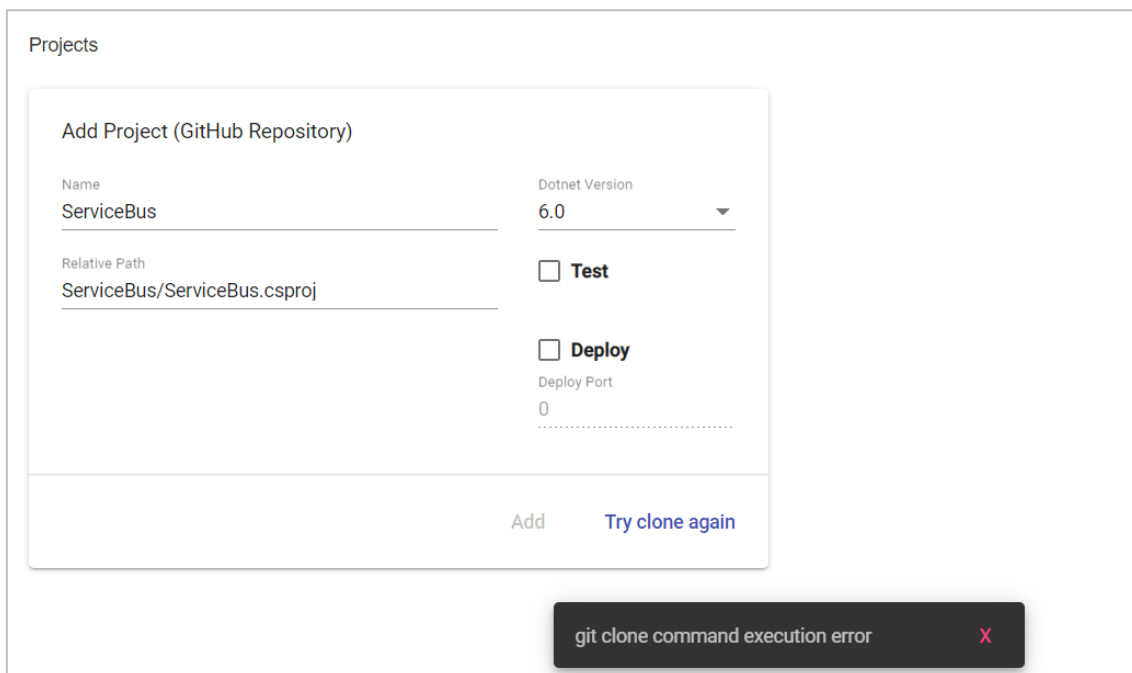
Let's say we delete a project manually from the database, but we forget to delete our working directory project folder.

We launch the application and try to add this project again.

After validating the project, it tries to clone it by executing a "git clone" command through the API.

At this moment, "git clone" command fails because there is already a folder for this project on our working directory and it's not empty.

Our API returns an error and this is what we see on our screen:



The screenshot shows a web application interface with a form titled "Add Project (GitHub Repository)". The form contains the following fields and controls:

- Name:** ServiceBus
- Dotnet Version:** 6.0 (dropdown menu)
- Relative Path:** ServiceBus/ServiceBus.csproj
- Test:** ☐
- Deploy:** ☐
- Deploy Port:** 0
- Buttons:** "Add" and "Try clone again"

Below the form, a dark error message box is displayed with the text "git clone command execution error" and a red "X" icon.

If we can't realize what caused the error, we have to go to our API console in order to look for any message which tells us what happened.

```
[19:23:55 INF] Executing command: git clone https://gh[REDACTED]@github.com
/rlrecalde/ServiceBus.git
fatal: destination path 'ServiceBus' already exists and is not an empty directory.
[19:23:55 ERR] Response code: 500; Body: {"Message":"git clone command execution error","StackTrace":null,"Data":[]}
```

Given that some git commands don't write to the standard output, there is no way to make our API to return that error message to us. That's why we have to look for the error message on the API console.

As a rule of thumb, every time we get an error message on our screen we go to the API console to see what's going on.

As you can see on the picture above, after cloning command fails, a "Try clone again" button appears on the "Add Project" panel. So, we remove the folder from our working directory and then click on that button.

My best practices

Even though I'm not a frontend developer, I've realized Angular is an object oriented framework in which we need to get rid of those old structured programming habits we used to have when we developed web pages with javascript.

We need to understand that a component is an object; and also, it's better and cleaner for our resulting code to use a multitier architecture.

So, in order to develop frontend with Angular, we need to think as a backend developer.

Inside `\src\app` folder, we will see the following folders: "bo", "components", "dto", "modules", "pipes" and "services".

They are layers.

"dto" layer: Typescript interfaces (POCO entities) for communicating to the API.

"bo" layer: Typescript interfaces and enumerations acting as business objects for our business logic.

"pipes" layer: Pipes are mappers; not only for the UI, but also for our business logic. For instance, we can see "status.pipe.ts" which is used for the UI and also "project-to-project-work-flow.pipe.ts" which is used for business purposes.

"services" layer: This is the more important layer. Here we have a sub-layer called "api". In this sub-layer, each service has the responsibility of communicating to the API. Each of them corresponds to an API Controller.

The other services have business logic responsibilities. We can think about them as different business classes that only have methods.