Tutorial 1 - SOPRA

Lucio Canepa Joel Cedric Schmidt Luke Benjamin Fohringer

2026-02-06

Table of contents

1 Introduction

The goal of SOPRA and particurlaly Milestone 1 is to get familiar with the basics of Full-Stack web development with the chosen frameworks.

This guide offers step-by-step tutorials to set-up your local machine for development and deploy it. Conceptual explanations and recommandations are also included.

What's a Web Application? - Main concepts

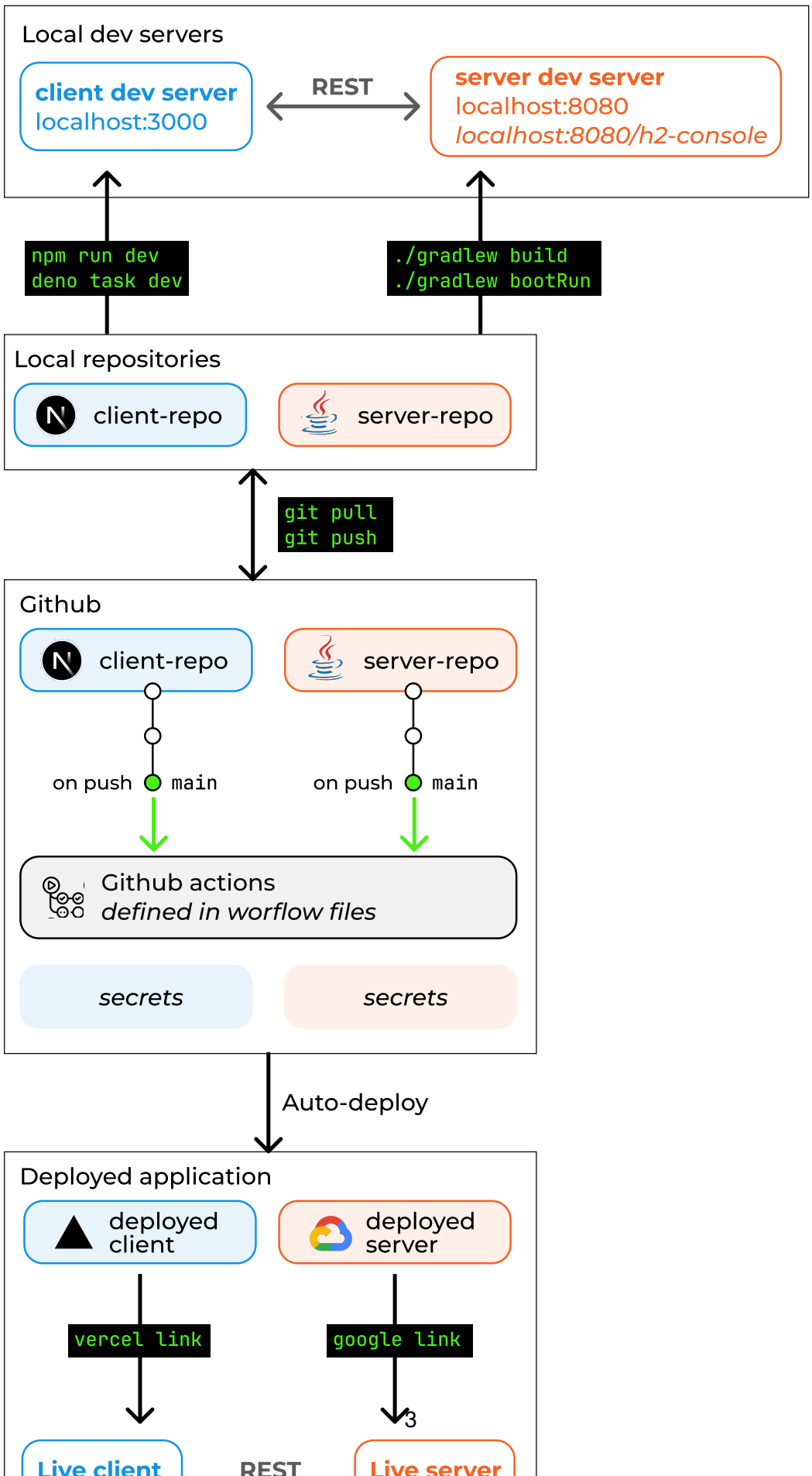
Web Application components
Web applications are usually built with 3 fundamentals components:

- **Client:** This is what the users see when visiting a web application. It's responsible to fetch the data and display it to the user in an interactive way. The client can interact with the data trough the **backend** and a defined API between them (in our and most cases

REST)

- **Backend (Server):** The backend has direct access to the database (can read and write) and offers specific access points (**endpoints**) to clients. It's responsible to manage the complexity of retrieving the needed data that clients request and check access permission.
- **Database:** The database store the actual data. There can be many types of databases and languages to interact with it. In this course, a semi-persistent database (only lives in memory, RAM) based on Postgress and accessible trough SQL is set-up.

Diagram


Local dev servers
client dev server
localhost:3000
REST
server dev server
localhost:8080
localhost:8080/h2-console
npm run dev
deno task dev
./gradlew build
./gradlew bootRun
Local repositories
client-repo
server-repo
git pull
git push
Github
client-repo
server-repo
on push
main
on push
main
Github actions
defined in worflow files
secrets
secrets
Auto-deploy
Deployed application
deployed client
deployed server
vercel link
google link
Live client
REST
Live server

REST
REST stands for REpresentational State Transfer and offers **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete) operations hitting a specific endpoint.
Endpoints
Endpoints are specific URI to describe a specifc resource to retrive: `/users/{id}`.
Used in combination with REST:

```
GET /users/{id}
header: Authorization token (a key used by the backend
to grant acces to the resource)
```

> **Database**
>
> The setup of the server repository in SOPRA already contains and spins up the in-memory database. This means that there is no external database to handle.

Deployment
The 2 repositories templates available for sopra are:

- sopra-fs26-template-client
- sopra-fs26-template-server

responsible for the client and server.
These repositories are hosted on Github. As a convention, the `main` branch represents the latest working instance (what should be deployed and accessible by the users). The 2 repositories are respectively deployed on:

- client on Vercel
- server (together with the in-memory database) on Google Cloud

This happens automatically when someone pushes code changes on the `main` branch: this is managed with Github Workflow to support continuous development (there is no need to manually deploy applications).

Development
We usually refer to 2 main environments:

- production: the one we just discussed and acutally run the application accessible to the users
- development: where developers work on their local machine, change and share code while implementing new features and fixing bugs.

Developers clone the repositories hosted on Github on their local machine, work on the code and push the code (they share the changes to Github – with the other team members). If it's the case, the automatic CI/CD pipeline defined by Github workflows take care of it and redeploys the application.
To run and test the application locally, you need:

- local application running with the client
- local application running wiht the backend (server)

Development flow

- clone application locally
- install needed dependencies to run it locally
- implement a new feature / fix a bug
- test locally that everything works correctly
- push the new code the remote (Github) usually on a feature branch
- once the code is tested and ready to be integrated into production, the feature branch gets merged into the `main` branch, triggering the deployment

2 What do you need before starting

To clone the code on your machine and run it locally, you need to set-up / have installed:

- git: version control system
- java 17+ (JDK) installed and `JAVA_HOME` environment variable set
- Github account
- ssh key to authenticate with Github easily

You will install other needed packages from set-up files provided by the repositories.

> **Warning**
>
> If you are on Windows, make sure to install WSL (Windows Subsystem for Linux) first.

> **Tip**
>
> If your are on a Mac, I highly recommend installing Homebrew as package manager:
>
> - You can install packages with: `brew install <package_name>`
> - You can install applications with: `brew install --cask <application_name>`

Setup ssh key

> **Note**
>
> If you are on Windows, first install Linux Subsystem for Windows (WSL) and then follow the instructions below.

In your terminal:

```
ssh-keygen -t ed25519 -C "<my_computer_name>"
```

It will prompt you for:

- choosing location (default is fine)
- choosing a passphrase (optional: can leave blank for no passphrase)

Then, add the ssh key to the ssh-agent:

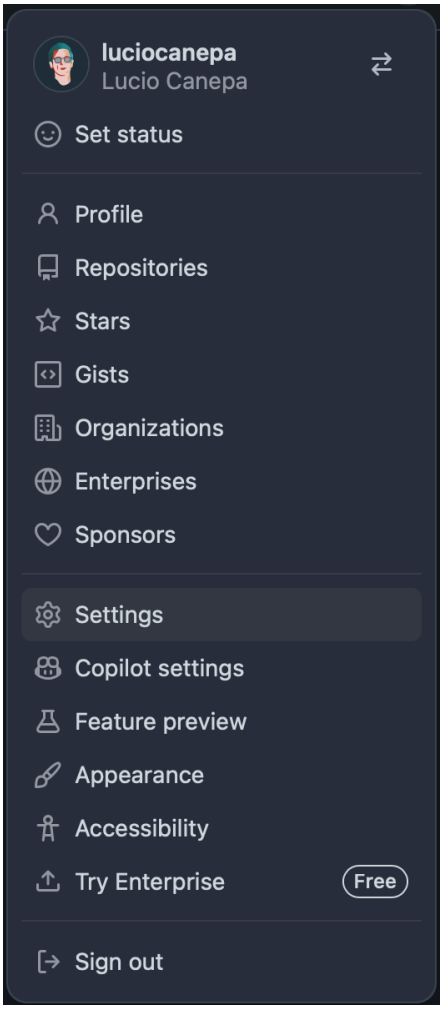
```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

You need the public key to add it to your Github account:

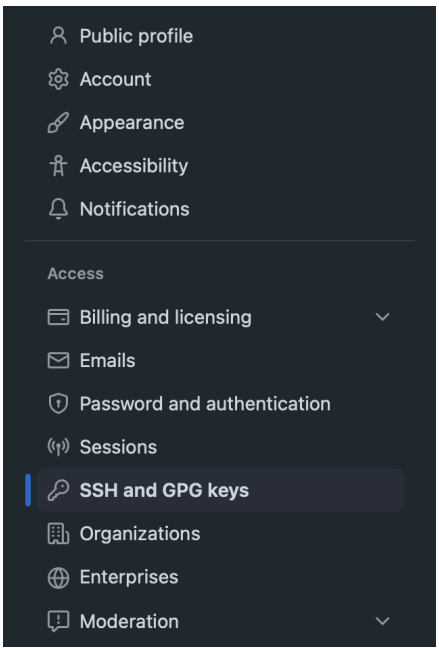
```
cat ~/.ssh/id_ed25519.pub
```

Link ssh key to Github account

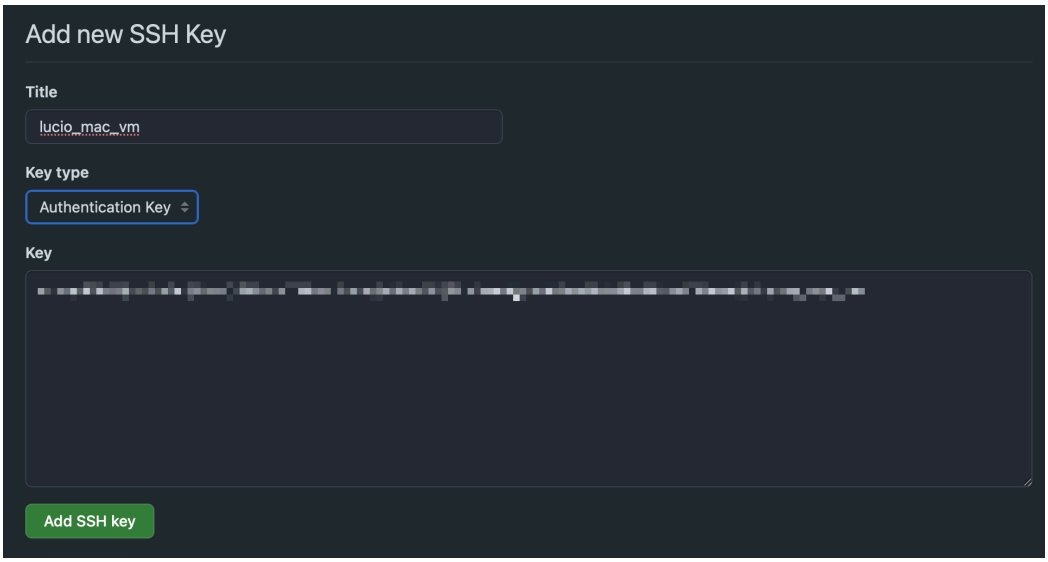
1. Go to your Github settings



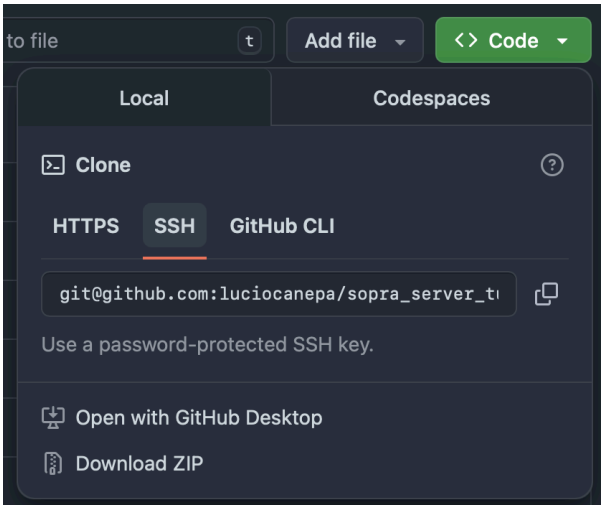
2. Go to “SSH and GPG keys”



3. Click on “New SSH key”, give it a title (e.g. “My Laptop”) and paste the public key you got from `cat ~/.ssh/id_ed25519.pub`



4. Once you added it, you can use ssh to clone repositories from Github:



Now you are able to act on remote repositories hosted on Github without entering your credentials every time. Always use the ssh url when cloning repositories from Github.

Install Java 17

Mac

If you are on a Mac, you can install Java 17 with Homebrew:

```
brew install --cask temurin@17
```

Run:

```
/usr/libexec/java_home -v 17
/Library/Java/JavaVirtualMachines/temurin-
 ↪  17.jdk/Contents/Home
```

to locate your Java 17 installation path.

Run:

```
which ${SHELL}
/bin/zsh
```

to find out which shell you are using.

Then, add the following line to your shell configuration file (`.zshrc` for zsh, `.bash_profile` or `.bashrc` for bash):

```
code ~/.zshrc
```

paste

```
export JAVA_HOME=$(/usr/libexec/java_home -v 17)
export PATH="$JAVA_HOME/bin:$PATH"
```

at the end of the file, save and close it.

Run `source ~/.zshrc` to reload the configuration file.

Linux / WSL

After making sure you have `git` and `curl` installed on your system:

```
sudo apt update
sudo apt install git curl
```

you can install Java 17 with:

```
sudo apt install openjdk-17-jdk
```

Depending on your package manager of choice.

Ensure you have Java 17 installed and configured correctly by running:

```
macvm@Macs-Virtual-Machine sopra_server_tutorial %
 ↪  java -version
openjdk version "17.0.17" 2025-10-21
OpenJDK Runtime Environment Temurin-17.0.17+10
 ↪  (build 17.0.17+10)
OpenJDK 64-Bit Server VM Temurin-17.0.17+10 (build
 ↪  17.0.17+10, mixed mode, sharing)
```

Install WSL on Windows

If you are using **Windows**, you need to install **WSL (Windows Subsystem for Linux)** before continuing with the Linux/WSL installation steps (Git, Curl, Java, …).

Tip

Save your work first: the installation may require a reboot.

1. Open **PowerShell as Administrator** Start Menu → search **PowerShell** → right-click → **Run as administrator**
2. Run:

```
wsl --install
```

3 Clone repositories & run code locally

Navigate to the folder where you want to store the project and clone the repository:

```
cd <path_to_your_folder>
git clone
 ↪  git@github.com:<user_name>/<repository_name>.git
```

where `<repository_name>` are the names of the server and client repositories you created from sopra templates.

Details

```
macvm@Macs-Virtual-Machine ~ % mkdir sopra
macvm@Macs-Virtual-Machine ~ % cd sopra
macvm@Macs-Virtual-Machine sopra % git clone
 ↪  git@github.com:<user_name>/sopra_server_tutorial.git
Cloning into 'sopra_server_tutorial'...
The authenticity of host 'github.com
 ↪  (140.82.121.4)' can't be established.
ED25519 key fingerprint is <your_SHA256_fingerprint>.
This key is not known by any other names.
Are you sure you want to continue connecting
 ↪  (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519)
 ↪  to the list of known hosts.
remote: Enumerating objects: 73, done.
remote: Counting objects: 100% (73/73), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 73 (delta 3), reused 72 (delta 3),
 ↪  pack-reused 0 (from 0)
Receiving objects: 100% (73/73), 71.61 KiB | 596.00
 ↪  KiB/s, done.
Resolving deltas: 100% (3/3), done.
macvm@Macs-Virtual-Machine sopra % git clone
 ↪  git@github.com:<user_name>/sopra_client_tutorial.git
Cloning into 'sopra_client_tutorial'...
remote: Enumerating objects: 52, done.
remote: Counting objects: 100% (52/52), done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 52 (delta 0), reused 52 (delta 0),
 ↪  pack-reused 0 (from 0)
Receiving objects: 100% (52/52), 120.59 KiB |
 ↪  620.00 KiB/s, done.
macvm@Macs-Virtual-Machine sopra % ls
sopra_client_tutorial   sopra_server_tutorial
```

> **Caution**
>
> Use the ssh url when cloning the repository.

At this point you can already look into the code, read the `README.md` file and run the project locally.

> **Tip**
>
> If you are using VSCode as code editor, you can open the folder directly from terminal with `code <repo_name>` after activating `Shell Command: Install 'code' command in PATH` from the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`).
> Or `code .` to open the current folder.

3.1 Client repository

> **Tip**
>
> If while `cd`-ing into a repo folder this message shows up:
>
> ```
> macvm@Macs-Virtual-Machine sopra % cd
>  ↪  sopra_server_tutorial
> direnv: error
>  ↪  /Users/macvm/sopra/sopra_server_tutorial/.envrc
>  ↪  is blocked. Run `direnv allow` to approve its
>  ↪  content
> ```
>
> run `direnv allow` to enable the automatic environment variable loading. If direnv is not installed, install it with your package manager of choice (or curl it).

Run `source setup.sh` as described in the `README.md` to install needed packages / dependencies.

Deno and npm (2 TS to JS runtimes) are available in the repository: use `npm run dev` to run the development server locally (at default port `localhost:3000`).

At `package.json` you can find available scripts:

```
"scripts": {
    "dev": "next dev --turbopack",
    "build": "next build",
    "start": "next start",
    "lint": "deno lint",
    "fmt": "deno fmt"
  },
```

The code for your app lives under `app/` folder: there you can find the entry point to the application at `app/page.tsx`. Try to change something to see it automatically update in the browser on save.

3.2 Server repository

To build the application, use the provided Gradle wrapper. You first need to grant permission to the executable:

```
chmod +x gradlew
./gradlew build
```

The first time you run this, the Gradle wrapper will download the necessary Gradle distribution.

If everything works correctly, you will see:

```
BUILD SUCCESSFUL in 2m 50s
7 actionable tasks: 7 executed
```

> **Note**
>
> As suggested in the `README.md`, you can build the application continuously with:
>
> ```
> ./gradlew build --continuous -xtest
> ```
>
> the two flags allow for:
>
> - `--continuous`: to automatically recompile the code on changes
> - `-xtest`: to skip running tests on every build

To run the server locally, use:

```
./gradlew bootRun
```

This runs on `localhost:8080` by default: there you will see `The application is running.`

At `localhost:8080/h2-console` you can access the H2 database console: Information needed to access the console are at `src/main/resources/application.properties`.

4 Git & Github workflows

4.1 Git short guide

To get started working with local / remote repositories and collaborate with your team members, it's important to get to know the most used git commands. Videos, the official documentation and LLMs (with caution) are a good starting point.

Here we selected a subset of commands we consider relevant for this course.

Fundamentals

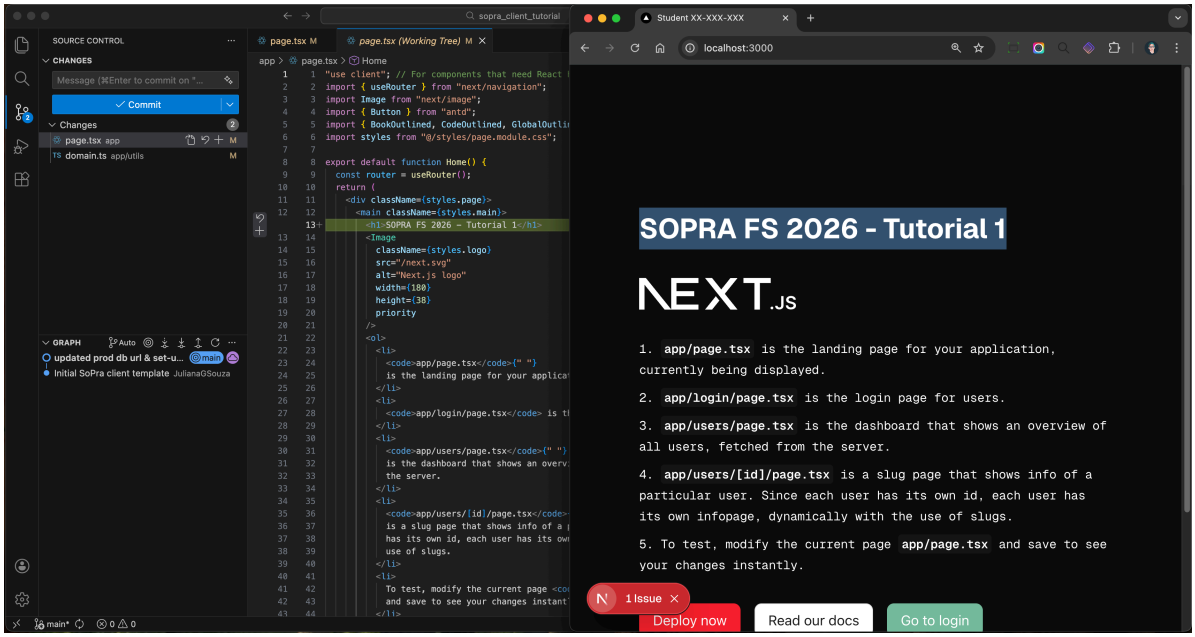
Git is a version control system based on branches that allows users to keep track of the file changes at a location. It allows to store incremental changes (`commits`) and to develop experimental features separately to the main code (`branch` and `merge`). ALl of this happens locally, on your machine. It's possible to connect your local repository to a remote origin (this happens automatically when you `clone` a repository from Github). Github simply acts as a second peer (the upstream) to whom you can `push` changes and from which you can `pull` changes (added by a team member).
The repositories you work on for Milestone 1 only have 1 branch, the `main` branch (standard entry point for an application).

- `git pull` tries to copy the changes of the branch you are currently on from the remote repository to your local one. This allows you to stay up-to-date with changes from other users.

Let's say you modified a file in your repository:

---

add title to main `page.tsx`



`git status` is a very useful command that gives you the snapshot (current state) of your branch:

- tells you on what branch you are and wheter it's up-to-date with the remote branch
- lists all the files that were: modified, added, removed
- lists all the changes that are: untracked, unstaged, staged

You can move a change to the *staging area* (prepare them for a commit) with:

- `git add <file_name>`
- `git add .` to add all changes at once

Run `git status` to see what happened

Caution

Before commiting the first time, ensure your `user.name` and `user.email` are properly set. In this way you can *sign* the commit: use your Github email and username to do so with:

```
macvm@Macs-Virtual-Machine
 ↪  sopra_client_tutorial % git config
 ↪  --local user.name "<username>"
macvm@Macs-Virtual-Machine
 ↪  sopra_client_tutorial % git config
 ↪  --local user.email "<email>"
macvm@Macs-Virtual-Machine
 ↪  sopra_client_tutorial % git config
 ↪  --local user.name
<username>
macvm@Macs-Virtual-Machine
 ↪  sopra_client_tutorial % git config
 ↪  --local user.email
<email>
macvm@Macs-Virtual-Machine
 ↪  sopra_client_tutorial %
```

You can choose beetween:

- the `--local` flag (recommended) that sets name and email for this specific repository (you need to do it both on the server and on the client repositories). This allows you to have multiple repositories on your computer linked to remotes where you are logged with different accounts / you want to sign your commits differently
- the `--global` flag: all the commits on your machine are going to be signed with name and email you set

With `commit -m "<commit_message>"` you can create a commit with its commit message (where you described the changes / reference the issue / bug it solves)

You can see your commit with `git log`.
You can push the commit(s) to the remote repository with:

- `git push` if the branch already exist on the remote
- `git push --set-upstream origin <new_branch_name>` to register a new local branch with your commit(s) on the remote repository

Branching

A branch is a reference (pointer) to a commit, usually used to develop a new feature, fix a bug, etc. while maintaining the main branch (often – and in this course – continously deployed) always working.

Access branches

- You can move your `HEAD` to an already existing branch with `git checkout <branch_name>`
- You can create a new branch pointing at current `HEAD` location with `git checkout -b <new_branch_name>`

You can `fetch` updates of a branch / branches in a repository with:

- `git fetch <branch-name>` gets you updates of that branch
- `git fetch --all` tells you what's new: new branch(es) has been created
- `git branch` lists all locally available branches (often useful to run together with `git fetch --all`)

Manage branches

Once your work in a branch is finished and you are ready to integrate your changes to the `main` branch (to deploy it). You want to `merge` your branch on the `main` branch:

- move to main: `git checkout main`
- merge your_branch to main: `git merge <your_branch>`
- this will create new commits *on top* of the `main` branch. Your branch and commits still exists at `git checkout <your_branch>`
- to update the remote repository with your changes: `git push`

Caution

before merging, always `pull` to get the latest changes on `main`

If you are interested in other ways to manage your branches and history, look into `rebase` and `cherrypick`.

Delete a branch

- You can delete a branch locally with `git branch -d <branch_name>`
- You can *publish* your deletion (delete the branch on the remote repository as well) with `git push origin --delete <branch_name>`

Recovery

If while working you realize you commited / pushed / lost something accidentally git offers some commands to rescue your code.

Reshape local history

If you didn't yet pushed your changes / commit to the remote repository, you are free to modify them as you wish:

- `git reset --soft HEAD~1` removes your last commit and leaves your changes in the staging area. This is useful if you want to add / remove something from that commit or change the commit message. (**Be careful:** make sure to use the `--soft` flag)
- `git rebase -i <commit_hash>` lets you modify interactively the commits from your `HEAD` to the commit the hash references to. This is useful to `squash` commits together, modify their commit messages or `drop` them.

When needed (*eg.* when rebasing), git opens an interactive shell. By default is vi. If you wish to change it, you can configure the behavior with:

```
git config --global core.editor "code --wait"
```

- `--global` or `--local` flags to change the editor everywhere or just for this repository
- the `--wait` flag is needed for window-based editors: it waits until you close the editor to proceed. This is not needed with terminal-based editors.

Recover a lost commit / branch

A useful command in this scenario is `git reflog`. It shows all the recent *places* (hashes) you `HEAD` has been pointing to: you can checkout to the desired hash, branch off and recover that snapshot.

Reshape remote (shared) history

As a general advice, you should **never** change the history of an already pushed commit / branch. This can cause a lot of troubles to your team members and prevent you to recover specific changes or have a good sense of the rationale behind a change.

In an ideal world, each commit has the right size: not too small, not too big – contains one specific change described by a clear commit message that explains what the commit does and references the issue / bug it implements / solve. Unfortunately we don't live in an ideal world and sometimes the history of a repository is not as clean as we wish.

If you want a clean history on the main branch / on specific feature branches, it's advised to:

- rebase the feature branch before merging on the main branch: many changes appear as a single commit and the remote branch stays untouched (preserving the full history)
- create a new *cleaned* branch to keep working on / to merge, but keeping the old branch for future reference / recovery.

You should decide as a team what you want to prioritise and what approach / rules to follow in this project.

Resolve conflicts

When collaborating on a git repository, more developers may modify the same file / functionality. This divergence creates conflict:

> git asks itself: what should I keep between the 2 options I see?

In most cases, git solves this conflicts automatically:

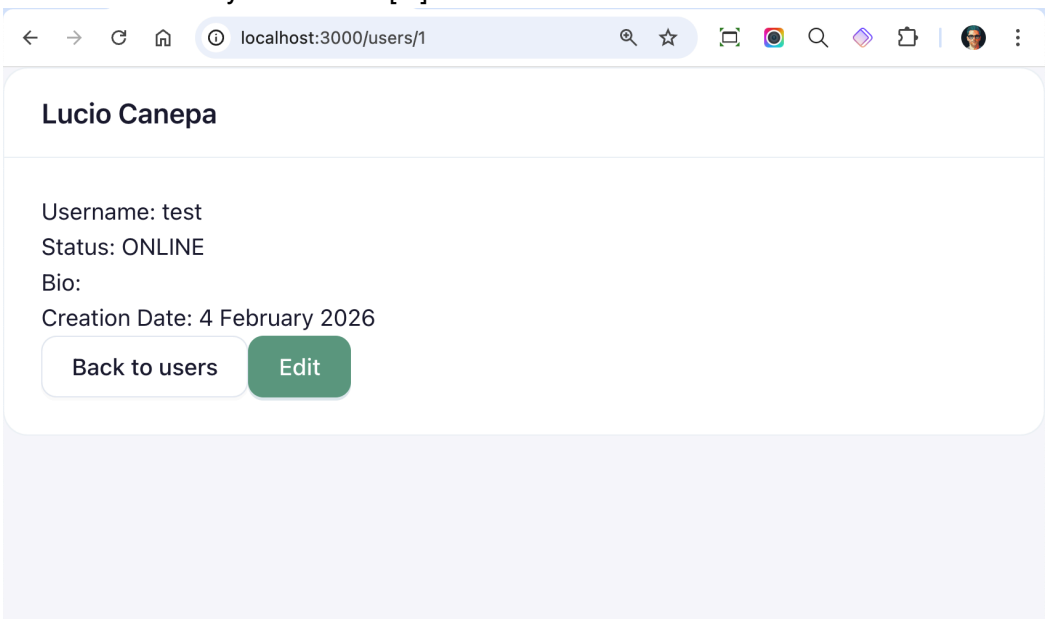
- *eg.* when pulling, if you set `git config pull.rebase true`, git tries automatically to move your changes on top of the pulled commits).
- when changes are on same file / functionality but not conflicting: can keep both of them

git will prompt you to resolve conflicts only when it wasn't able on its own to decide what to keep and what to ignore. This often happens when you merge a `feature` branch into your `main` branch. Most code editors (VS code *eg.*) have built-in conflicts resolution tools we suggest to use.

> **Demonstration**

Merge without conflicts

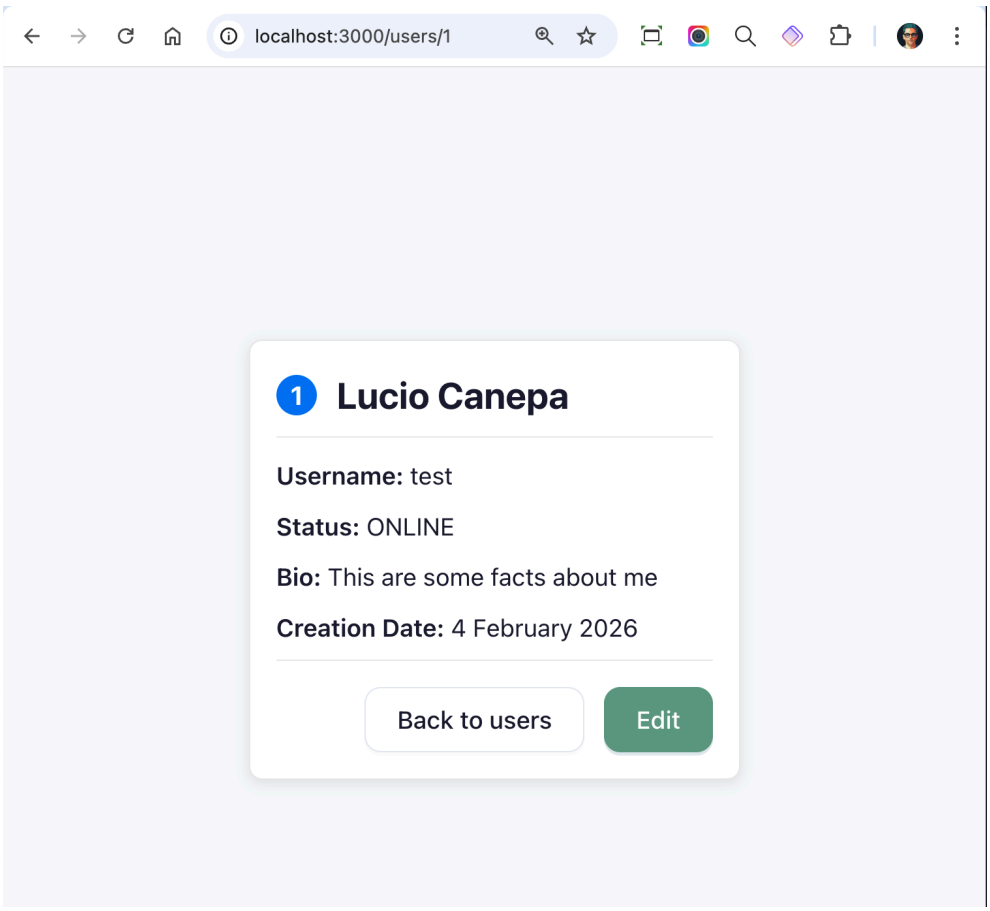
We want to modify how users/[id] looks



For this let's create a branch to work on: `#101_restyle_user`

```
macvm@Macs-Virtual-Machine sopra_client_tutorial % git checkout -b "#101_restyle_user"
Switched to a new branch '#101_restyle_user'
macvm@Macs-Virtual-Machine sopra_client_tutorial % git status
On branch #101_restyle_user
nothing to commit, working tree clean
macvm@Macs-Virtual-Machine sopra_client_tutorial %
```

There we modify the code and get:



2 new files and 1 modified - we can commit and push (on `#101_restyle_user`)

```
macvm@Macs-Virtual-Machine sopra_client_tutorial % git status
On branch #101_restyle_user
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   app/users/[id]/page.tsx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        app/components/CustomCardComponent.module.scss
        app/components/CustomCardComponent.tsx

no changes added to commit (use "git add" and/or "git commit -a")
macvm@Macs-Virtual-Machine sopra_client_tutorial % git add .
macvm@Macs-Virtual-Machine sopra_client_tutorial % git commit -m "new CustomCardComponent - s
tyled users/[id] page"
[#101_restyle_user d90b2e7] new CustomCardComponent - styled users/[id] page
 3 files changed, 137 insertions(+), 20 deletions(-)
 create mode 100644 app/components/CustomCardComponent.module.scss
 create mode 100644 app/components/CustomCardComponent.tsx
macvm@Macs-Virtual-Machine sopra_client_tutorial % git push -u origin "#101_restyle_user"
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 5 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.71 KiB | 1.71 MiB/s, done.
Total 9 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for '#101_restyle_user' on GitHub by visiting:
remote:      https://github.com/luciocanepa/sopra_client_tutorial/pull/new/%23101_restyle_use
r
remote:
To github.com:luciocanepa/sopra_client_tutorial.git
 * [new branch]      #101_restyle_user → #101_restyle_user
branch '#101_restyle_user' set up to track 'origin/#101_restyle_user'.
macvm@Macs-Virtual-Machine sopra_client_tutorial %
```

when we checkout to `main` and merge everything works: git was able to resolve conflicts on its own

```
macvm@Macs-Virtual-Machine sopra_client_tutorial % git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
macvm@Macs-Virtual-Machine sopra_client_tutorial % git merge "#101_restyle_user"
Updating 92f92db..d90b2e7
Fast-forward
 app/components/CustomCardComponent.module.scss | 55 +++++++++++++++++++++++++++
 app/components/CustomCardComponent.tsx         | 23 ++++++++++++
 app/users/[id]/page.tsx                        | 79 +++++++++++++++++++++++++++++----------
 3 files changed, 137 insertions(+), 20 deletions(-)
 create mode 100644 app/components/CustomCardComponent.module.scss
 create mode 100644 app/components/CustomCardComponent.tsx
macvm@Macs-Virtual-Machine sopra_client_tutorial %
```

The new commit is added to the history of `main`

```
commit d90b2e7bf1defeba51966fb059deb7f2f41bf5ed (HEAD -> main, origin/#101_restyle_user, #101
_restyle_user)
Author: username <email>
Date:   Wed Feb 4 17:16:26 2026 +0100

    new CustomCardComponent - styled users/[id] page

commit 92f92db27edb30b308d89c9c8e8a65353debe70a (origin/main, origin/HEAD)
Author: luciocanepa <lucio.canepa.6@gmail.com>
Date:   Sun Feb 1 12:03:37 2026 +0100

    Implementation of Bearer token

commit 061fdd9e15b9f24d9d7c05cf2b4e42c279fde28e
Author: luciocanepa <lucio.canepa.6@gmail.com>
Date:   Sat Jan 31 22:03:33 2026 +0100

    Users/id & users/id/edit pages
```

Merge with conflicts

Let's modify the `userHeader()` function on `#101_restyle_user`

```
41  41  const userHeader = () => {
42  --    const nameToDisplay = user?.name ? user?.name : user?.username;
    42+   // MODIFIED LINE: Logic changed to prioritize Username over Name
    43+   const nameToDisplay = user?.username || user?.name || "Unknown User";
43  44    return (
44  --      <>
45  --        <p>{user?.id}</p>
46  --        <h2>{nameToDisplay}</h2>
47  --      </>
    45+     <div className="header-container">
    46+       {/* MODIFIED LINE: Wrapped ID in a span and changed tag to h3 */}
    47+       <span>ID: {user?.id}</span>
    48+       <h3>{nameToDisplay}</h3>
    49+     </div>
48  50    );
49  51  };
```

Commit and push (on `#101_restyle_user`)

```
macvm@Macs-Virtual-Machine sopra_client_tutorial % git status
On branch #101_restyle_user
Your branch is up to date with 'origin/#101_restyle_user'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   app/users/[id]/page.tsx

no changes added to commit (use "git add" and/or "git commit -a")
macvm@Macs-Virtual-Machine sopra_client_tutorial % git add .
macvm@Macs-Virtual-Machine sopra_client_tutorial % git commit -m "new logic - will trigger co
nflict"
[#101_restyle_user d91475b] new logic - will trigger conflict
 1 file changed, 7 insertions(+), 5 deletions(-)
macvm@Macs-Virtual-Machine sopra_client_tutorial % git push
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 5 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 735 bytes | 735.00 KiB/s, done.
Total 6 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To github.com:luciocanepa/sopra_client_tutorial.git
   d90b2e7..d91475b  #101_restyle_user → #101_restyle_user
macvm@Macs-Virtual-Machine sopra_client_tutorial %
```

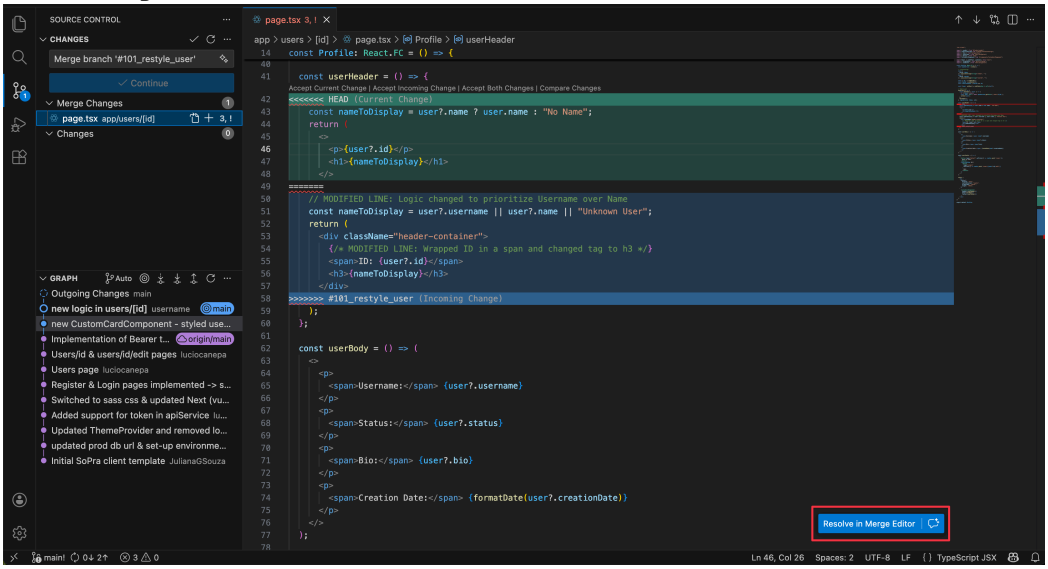
Let's modify the same function `userHeader()` on `main` in a different way - commit

```
41  41  const userHeader = () => {
42  --    const nameToDisplay = user?.name ? user?.name : user?.username;
    42+   const nameToDisplay = user?.name ? user.name : "No Name";
43  43    return (
44  44      <>
45  45        <p>{user?.id}</p>
46  --        <h2>{nameToDisplay}</h2>
    46+       <h1>{nameToDisplay}</h1>
47  47      </>
48  48    );
49  49  };
```

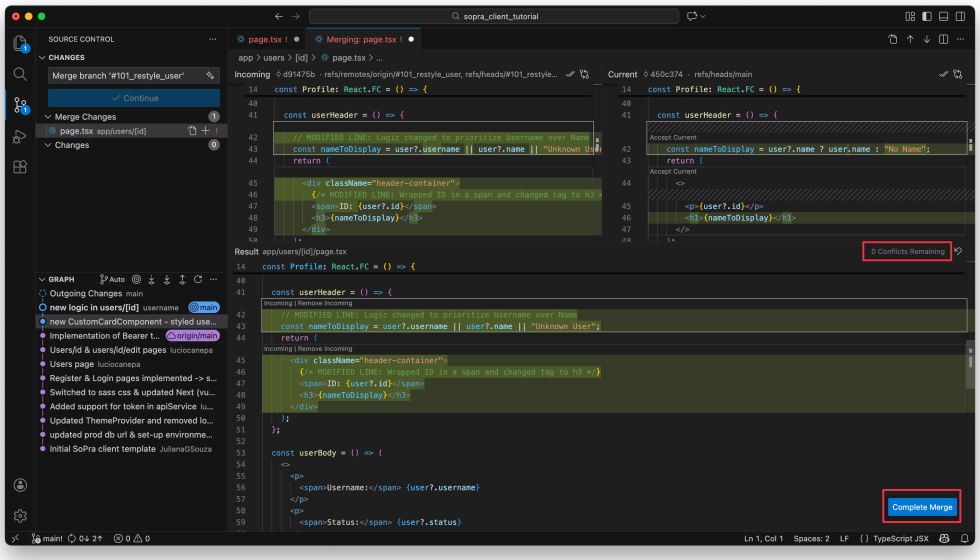
Try to merge `#101_restyle_user` on `main` → conflict

```
macvm@Macs-Virtual-Machine sopra_client_tutorial % git merge "#101_restyle_user"
Auto-merging app/users/[id]/page.tsx
CONFLICT (content): Merge conflict in app/users/[id]/page.tsx
Automatic merge failed; fix conflicts and then commit the result.
macvm@Macs-Virtual-Machine sopra_client_tutorial %
```

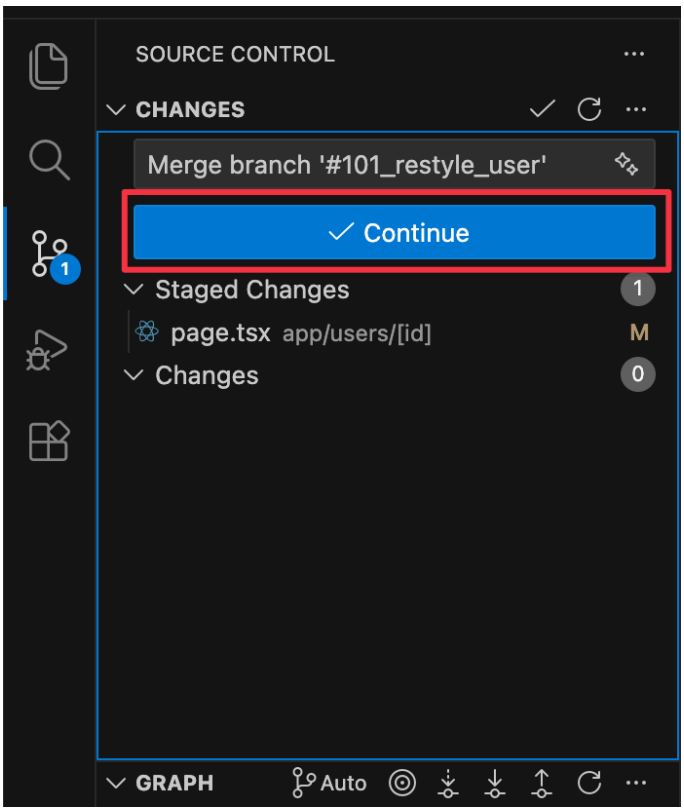
In VS code (under the git tab) you can see merge conflicts - click on `Resolve in Merge Editor`



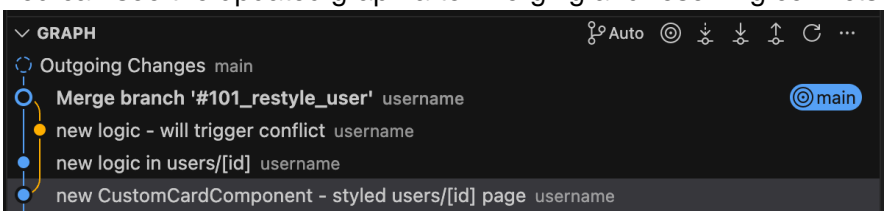
Here you can choose what to keep and what to discard (or manually update the file to a new resolution) - click `Complete Merge` when you are done (0 conflicts)



You can click `continue` to proceed with the `merge` with conflicts resolved



You can see the updated graph after merging and resolving conflicts



Best practices

Commits

- keep your commits small / reasonable size: each commit should implement and be responsible of a specific change (add a feature, solve a bug)
- give a title (short description) to your commit messages, then explain what you did and why. If possible, reference to an issue / bug number the commit is about.

Branches

- name your branches in a clear way: decide with your team what is the criteria for naming branches:
  - camelCase, snake_case, etc.
  - just issue number / issue number + name
  - naming convention when a feature branch is checked out

General

- always `pull` before you start working (`fetch` as well)
- always `pull` before you `push`

4.2 Github workflows

The SOPRA project relies on **CI/CD** (Continuous Integration / Continuous Deployment) to deploy (publish) the changes from the repositories hosted on Github to:

- Vercel for the client
- Google Cloud for the server

Instead of manually testing and deploying your code, **Workflows** are defined using GitHub Actions. These workflows are defined in YAML files located in `.github/workflows/` inside your repository. They define when they should run and allow granular and automated control on the deployment.

Environment Variables & Secrets

Applications require configuration that changes depending on where the code is running (e.g., a database URL that is `localhost` on your laptop but a Google Cloud URL in production) or sensitive data (passwords, API keys).
Locally (`.env`): On your machine, you store these in a `.env` file. The repository already includes a `.gitignore` that prevents this file from being uploaded to GitHub.

> **Security Risk**
>
> Never commit your `.env` file or hardcode API keys/passwords in your source code. If you push a key to a public repository, it is compromised immediately.

Remotely (GitHub Secrets): To let GitHub Actions access these values (to login to Google Cloud or Vercel), you must store them in the repository settings under `Settings > Secrets and variables > Actions`.

Workflow example (client)

Workflow files are usually intuitive enough to read and unsderstand. If we take the client deployment to Vercel as an example (`.github/workflows/verceldeployment.yml`):

```yaml
name: Deploy to Vercel

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repo
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: "18"

      - name: Install Dependencies
        run: npm install

      - name: Deploy to Vercel
        run: npx vercel --prod --token ${{
 ↪  secrets.VERCEL_TOKEN }} --confirm
        env:
          VERCEL_ORG_ID: ${{ secrets.VERCEL_ORG_ID
 ↪  }}
          VERCEL_PROJECT_ID: ${{
 ↪  secrets.VERCEL_PROJECT_ID }}
```

When new code is pushed to the `main` branch, 1 jobs is runned:

- defines the type of job (deploy) and the environment (latest Ubuntu)
- defined the steps: first it install needed resources
- then – in this case – reaches out to Vercel to deploy the frontend application
- to do so it needs environmnet variables (secrets) to connect and authenticate to Vercel. This secrets are defined in the repository settings.

When deploying you need to make sure that the Environment variables are set correctly in the repository settings in Github. You can add / modify the content of these files to control their behavior. *eg.* you might want to deploy a staging environment (`develop` branch) and test your changes in a production-like environment without making them accessible in production (`main`).

5 Set-up Google Cloud and Vercel & deploy your project

To make the application available, you need to deploy your changes:

Server repository

`main.yml` file runs 2 jobs when pushed on `main`:

- test (optional for now): runs the Jtest on the packaged version and make the results available at SonarQube
- deploy: deploys the application to the Google Cloud instance set-up as described below.

`dockerize.yml` creates a docker container and register it. This is optional for now, but will be needed later: a dedicated section explains what docker is and how to set it up.

Client repository

`verceldeployment.yml` file deploys the frontend code to vercel when pushed on `main`

`dockerize.yml` does exactly the same, but for the client (optional for now).

`sonarcloud.yml` runs an analysis (scan) of the code and publish the resulsts to SonarQube (optional for now)

5.1 Set-up Google Cloud and link it to the server repository

To set up the backend infrastructure on Google Cloud, the following services and entities must be successfully configured and enabled (as illustrated in the step-by-step tutorials):

Core Infrastructure

- GCP Project: A dedicated project container named sopra-fs26-[lastname]-[firstname]-server.
- App Engine Application: The specific environment where the Java/Spring Boot server resides, localized in the europe-west6 region.

Identity & Access Management (IAM)

- Service Account: A virtual identity used by GitHub Actions to interact with your Google Cloud resources.
- Editor Role: The specific permission level granted to the Service Account to allow resource modification.
- Service Account Key: A generated JSON key that serves as the “password” for GitHub to log into your Google Cloud project.

Cloud APIs (Must be “Enabled”)

- Cloud Build API: Handles the building of your application code into a deployable format.
- App Engine Admin API: Allows external tools (like GitHub Actions) to manage and deploy versions of your App Engine service.

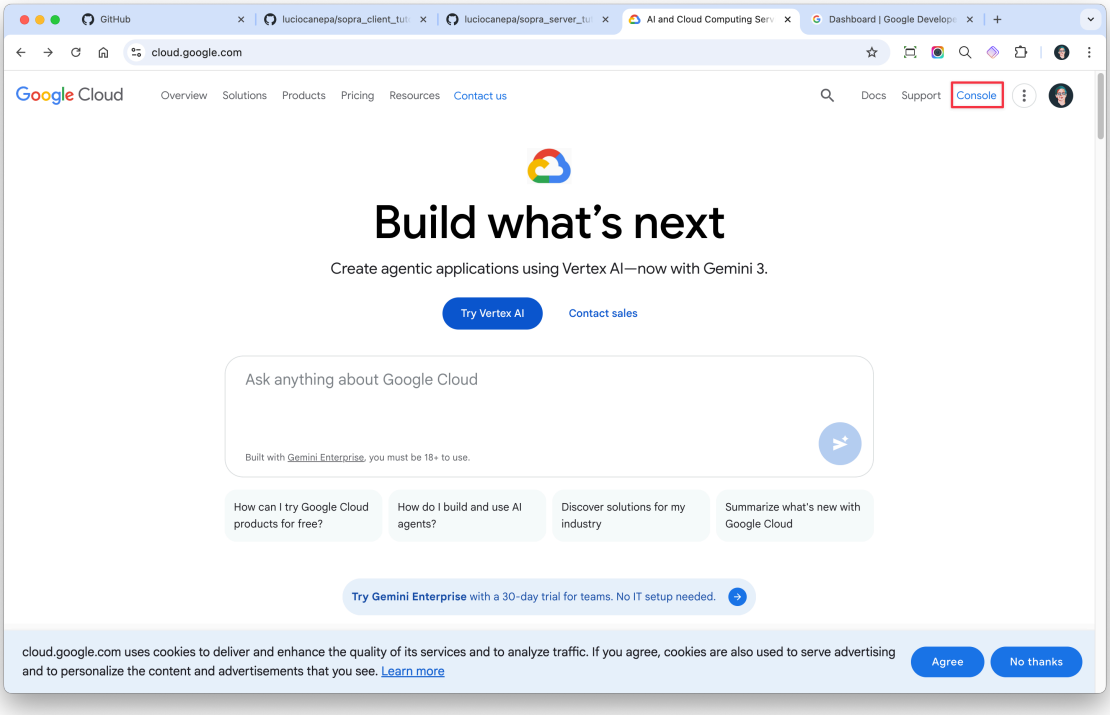
Secrets Management

- `GCP_SERVICE_CREDENTIALS`: The GitHub Actions secret containing the JSON key, which bridges the link between your code repository and the Google Cloud entities listed above.

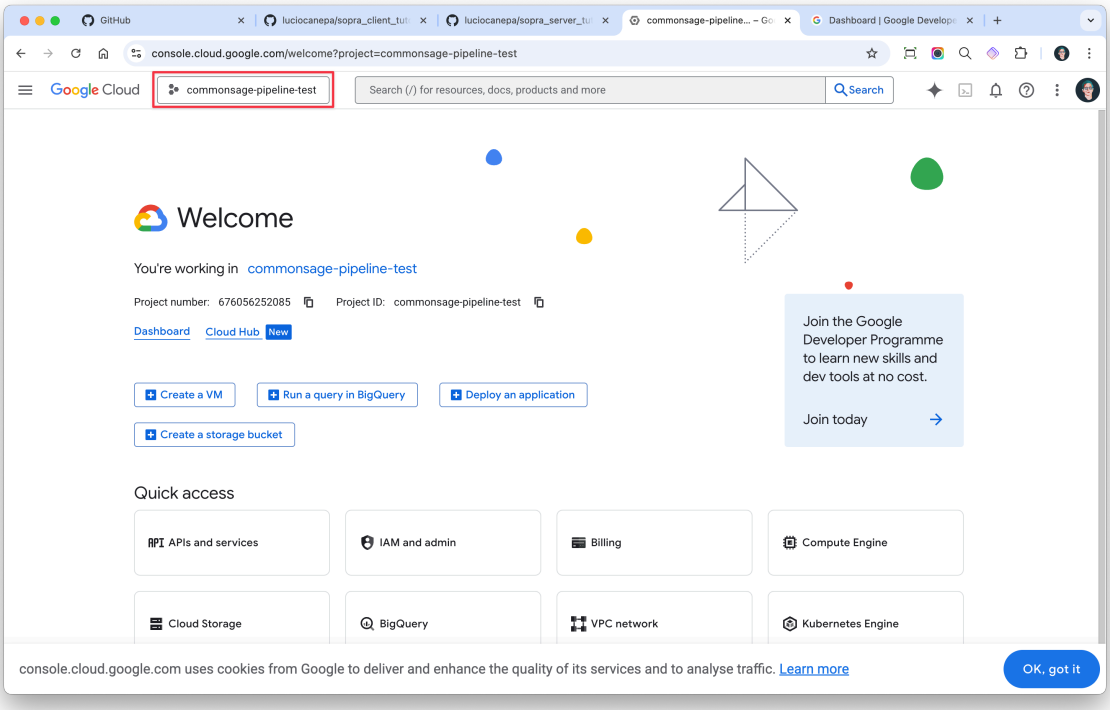
Set-up Google Cloud

Navigate to cloud.google.com and create an account / login

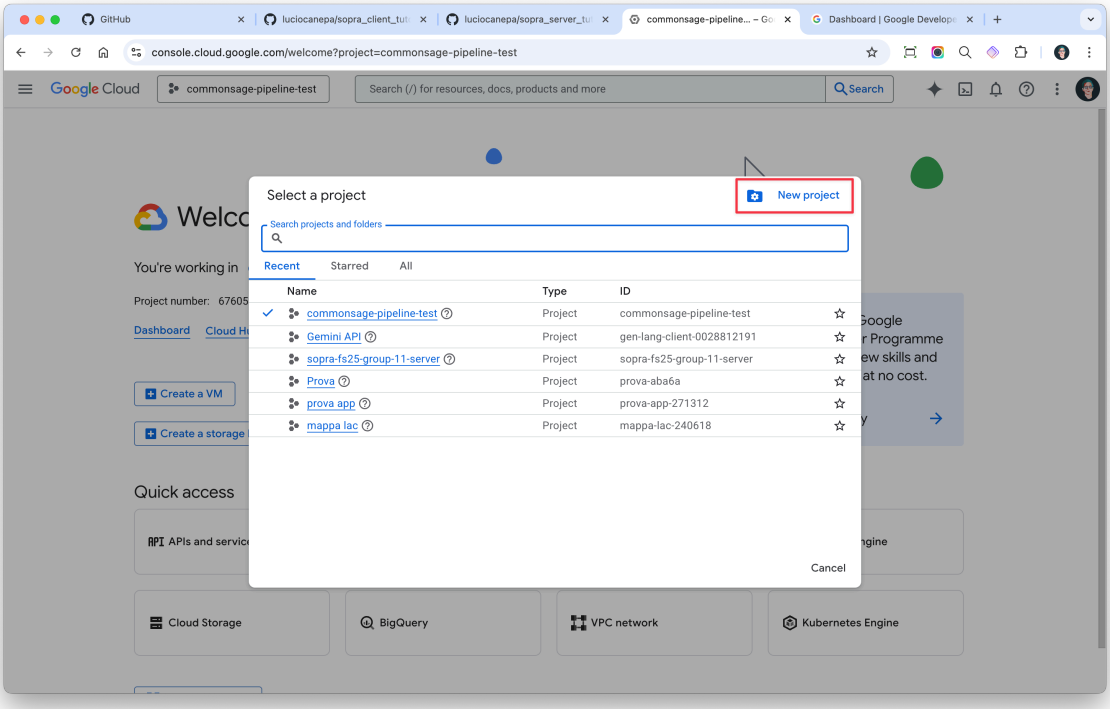
Click on `console`



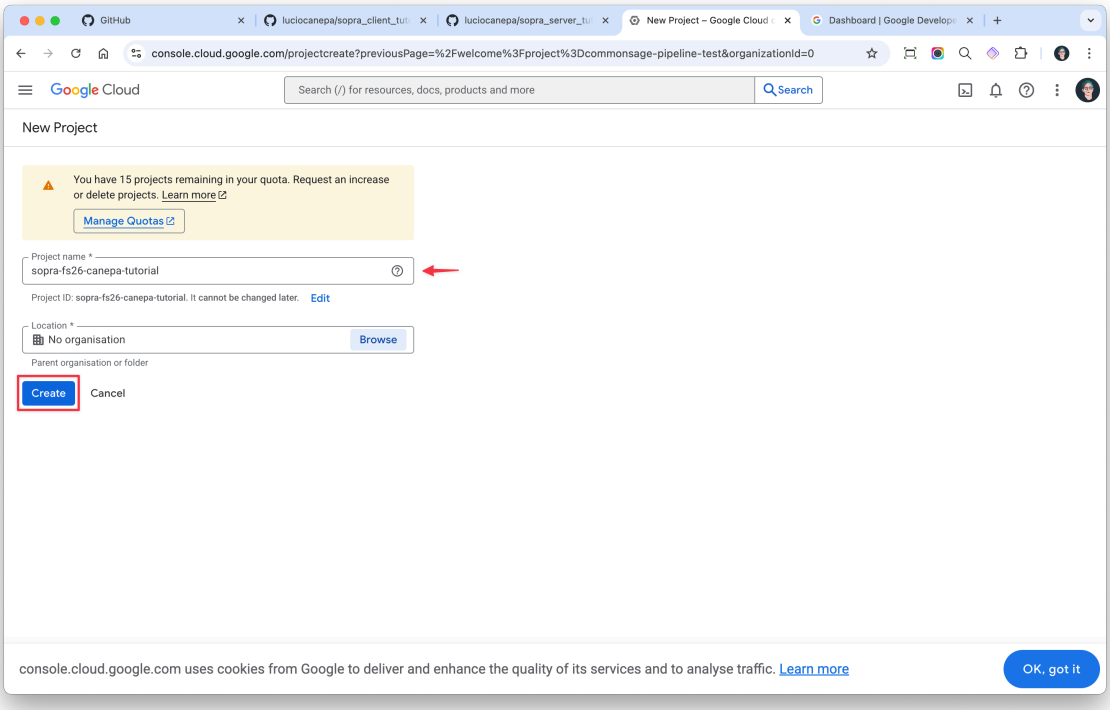
Click on the project selection



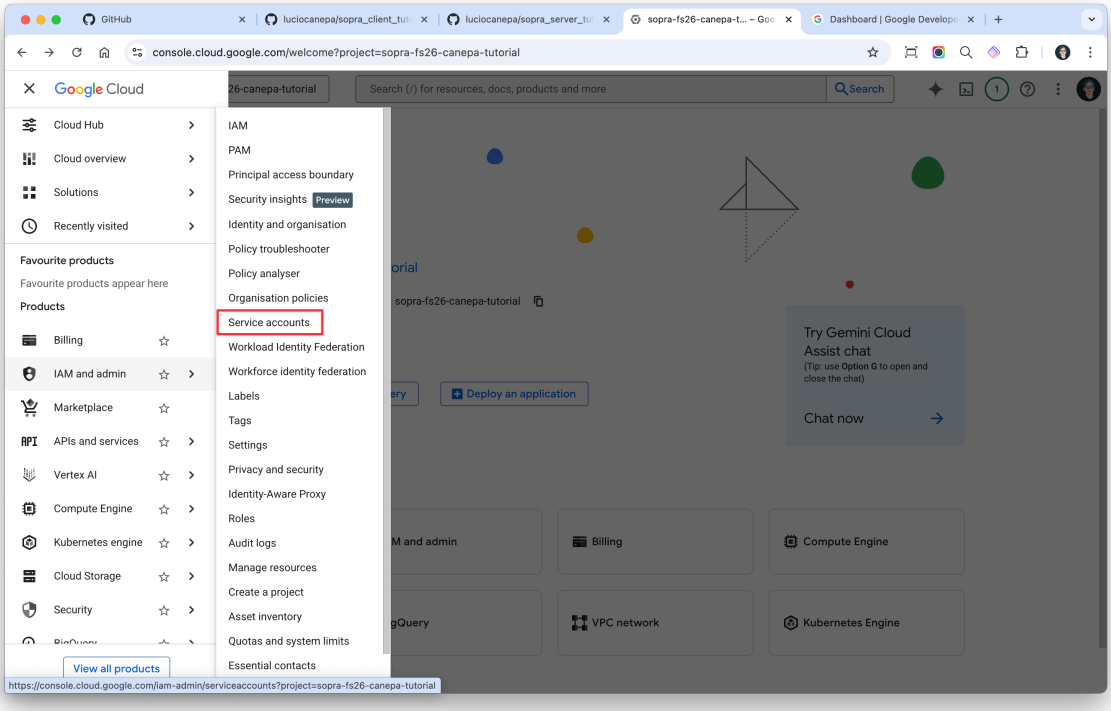
Create new project



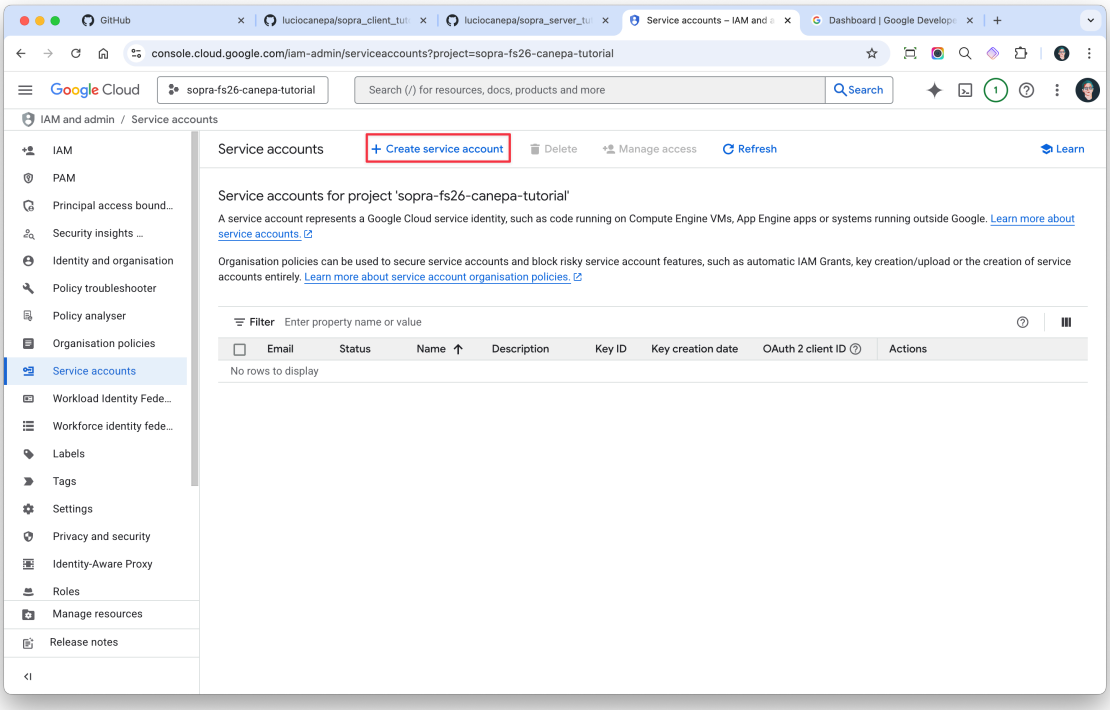
Name it according to specification:
`sopra-fs26-lastname-firstname-server` & click `create`



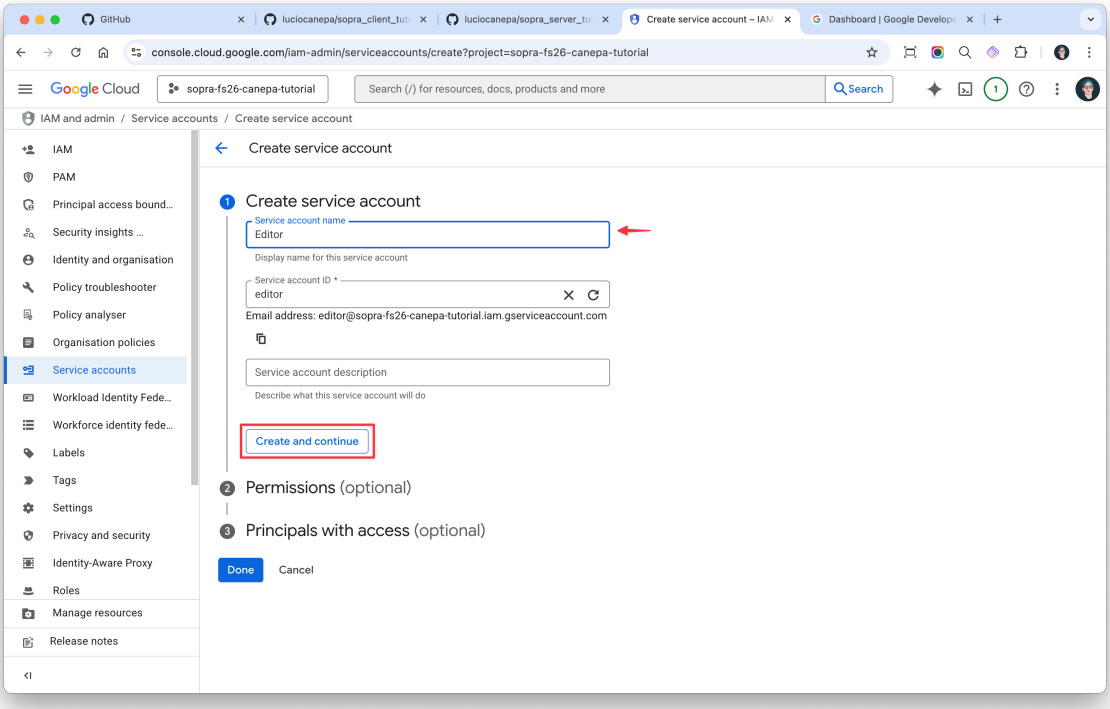
Navigate to `IAM and admin` < `Service accounts`



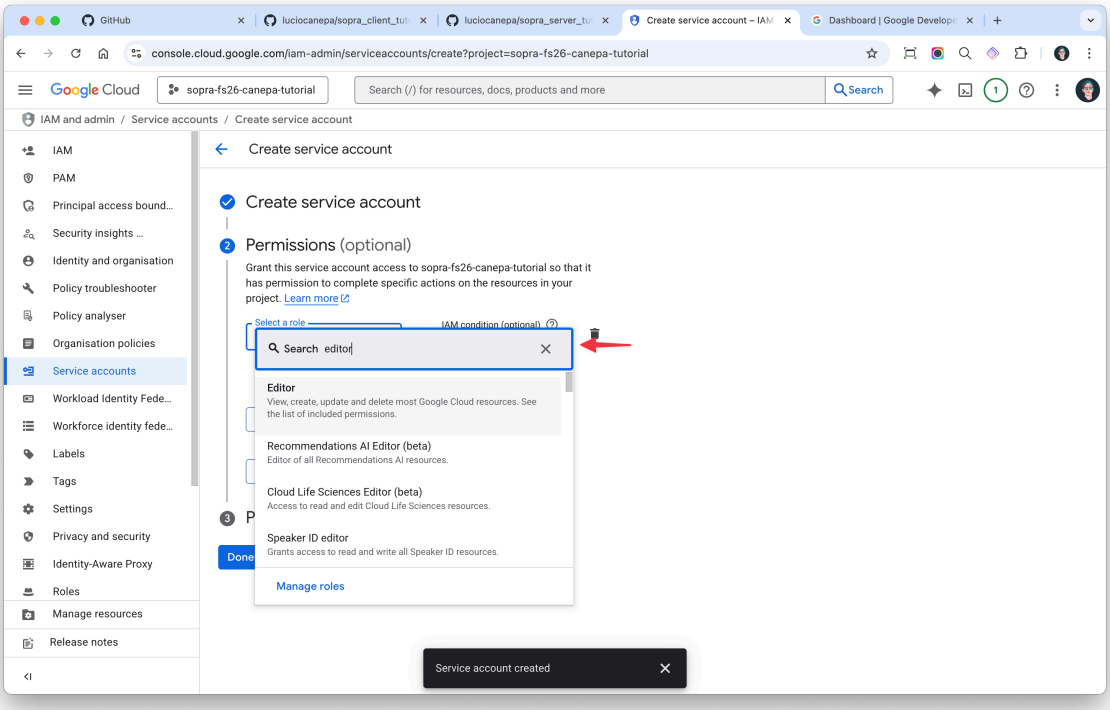
Create a new service account



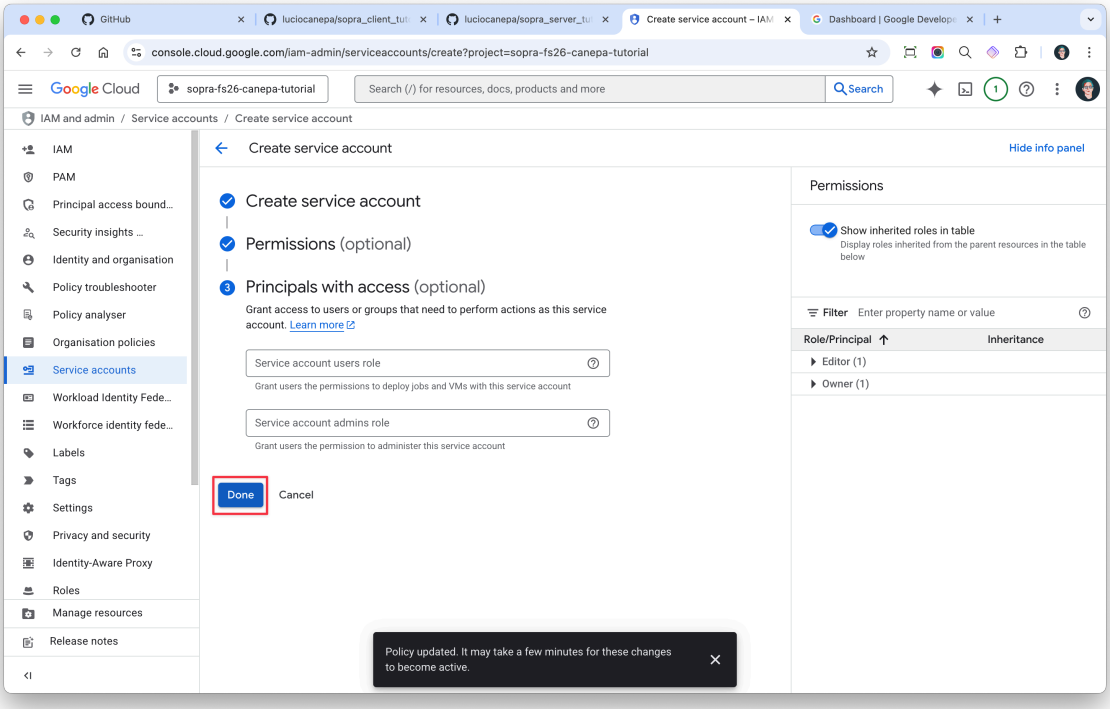
Name it as you wish (it will grant editor access) & click `Create and continue`



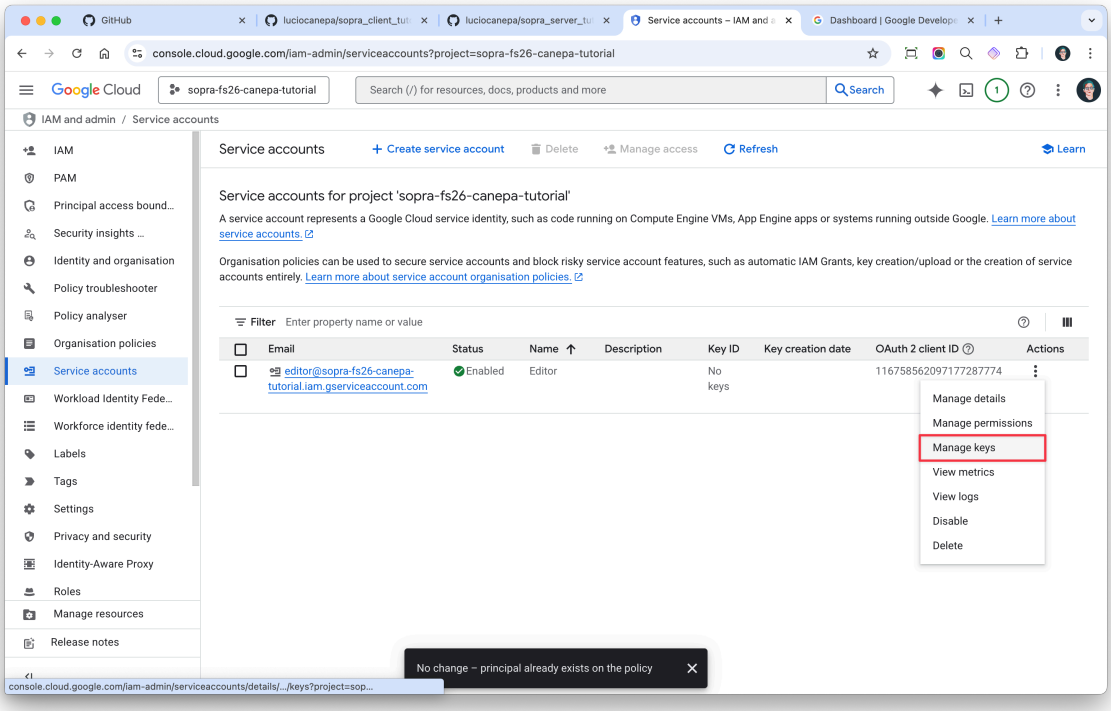
add editor role in step 2 - permission (you can search for it) & click `Continue`



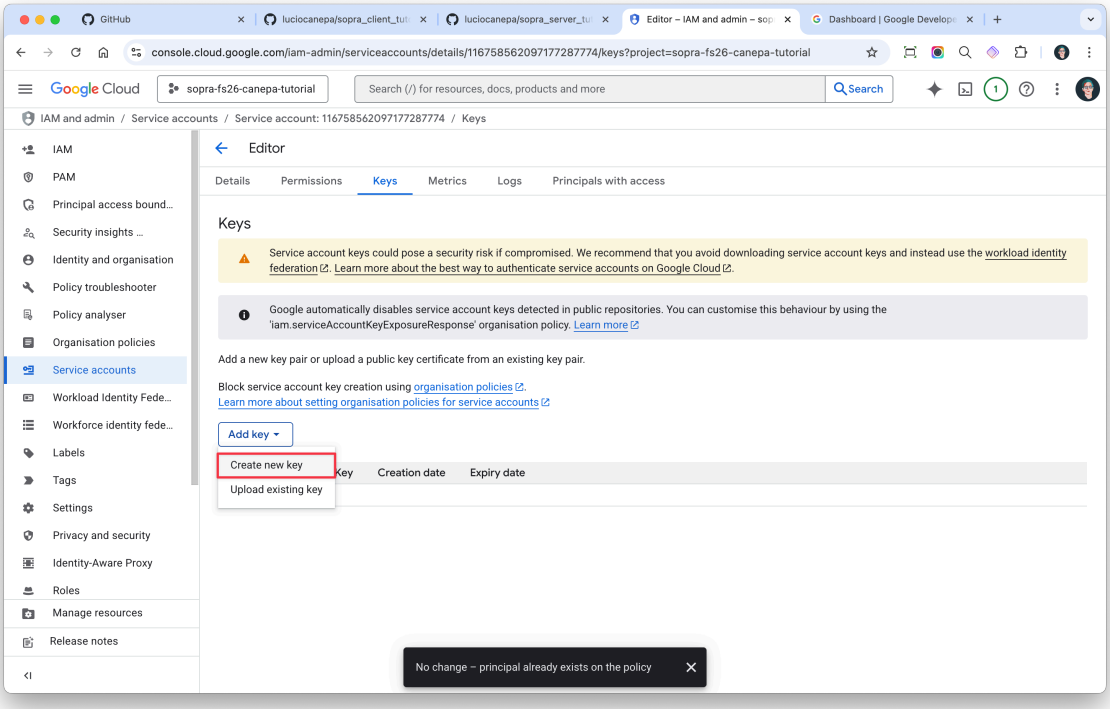
Click `done`



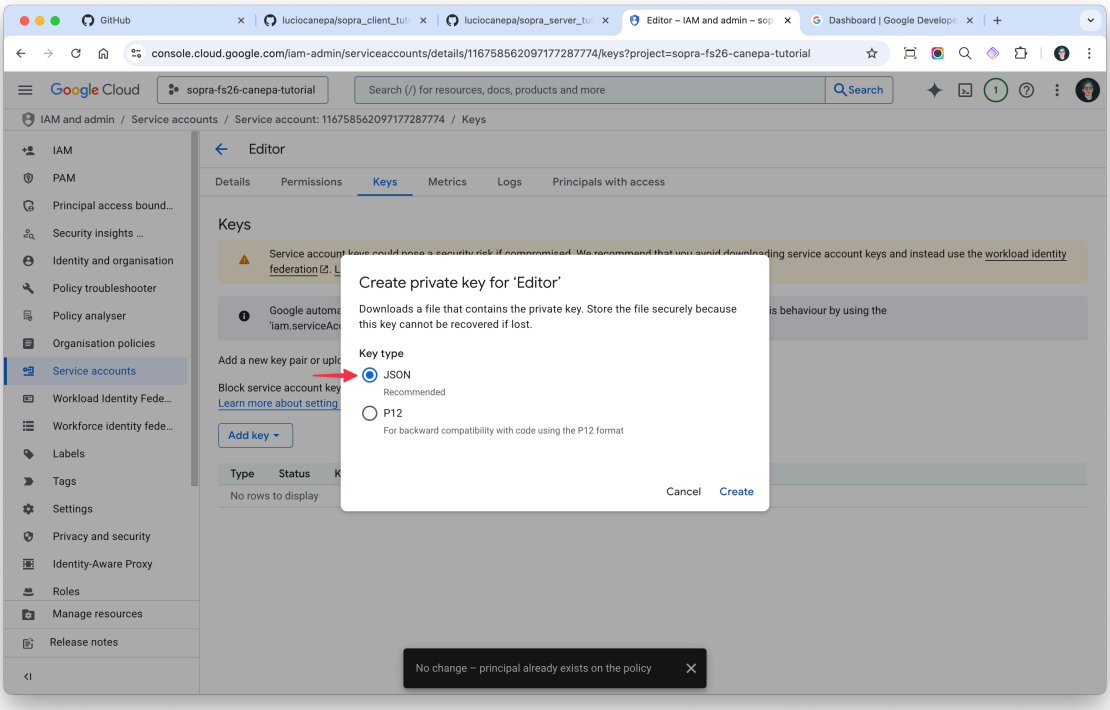
Next to the created service account, select more > `Manage keys`



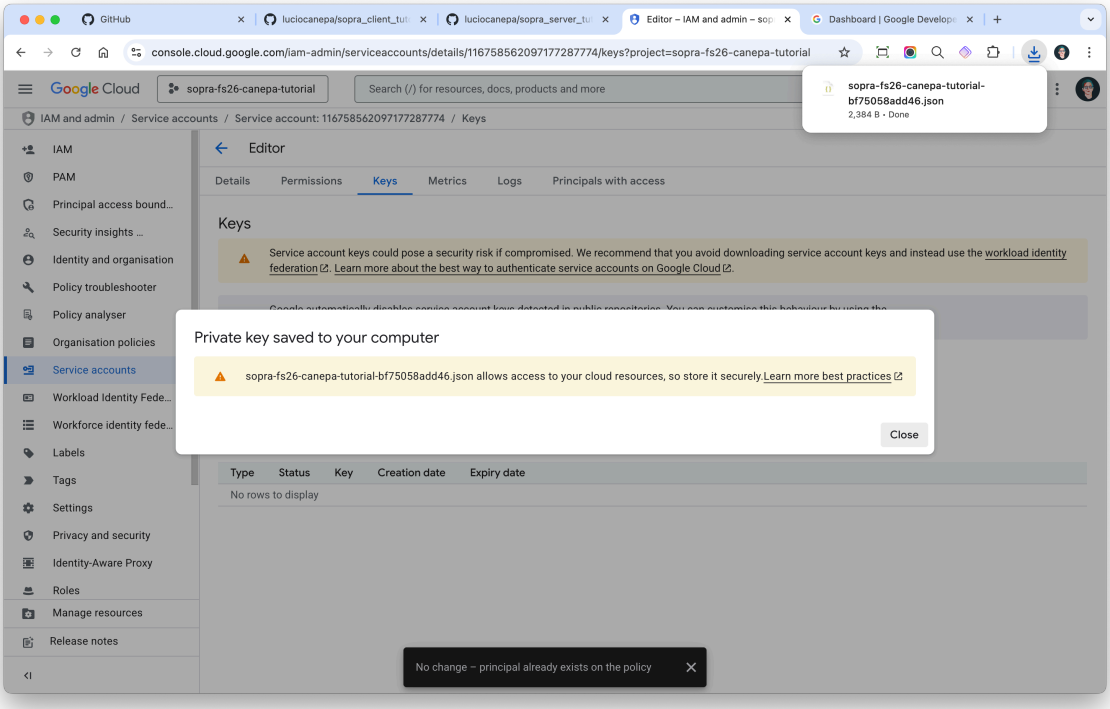
Add a new key



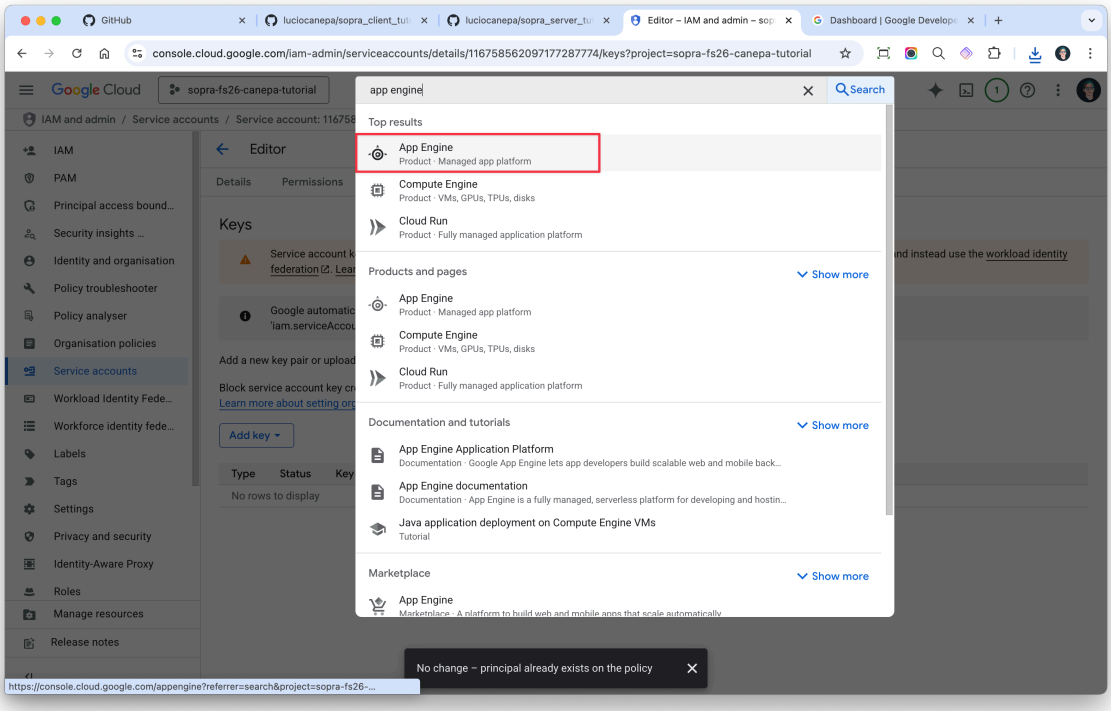
in JSON format (default)



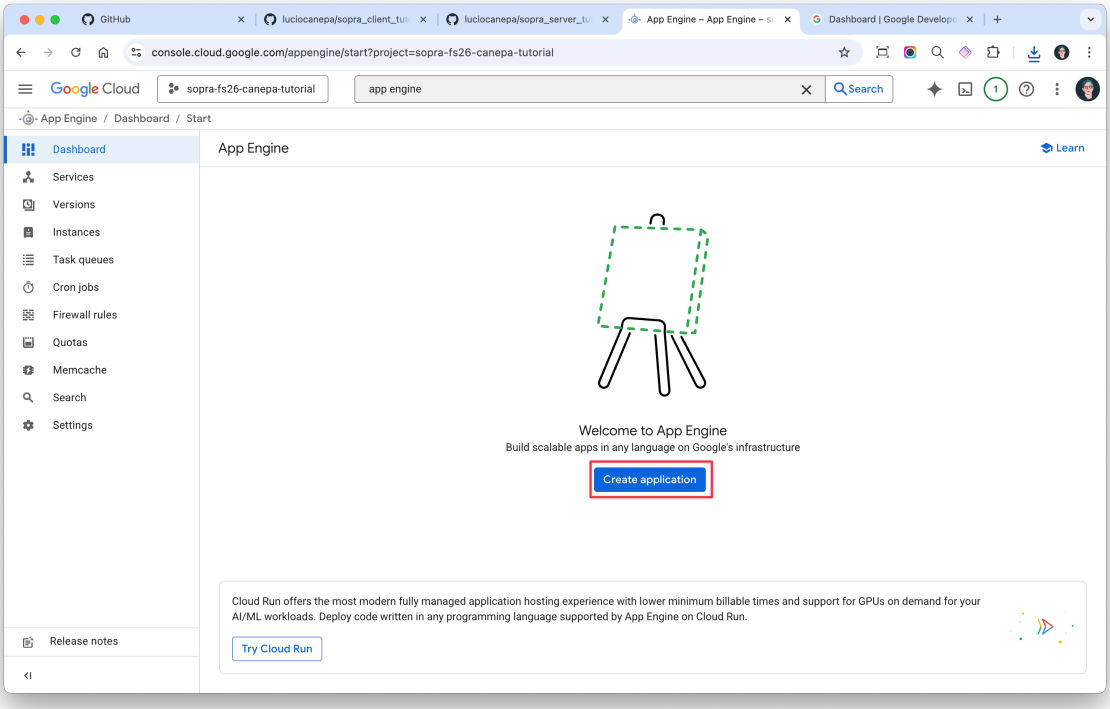
Key creation confirmation screen - the key as JSON file is saved on your computer



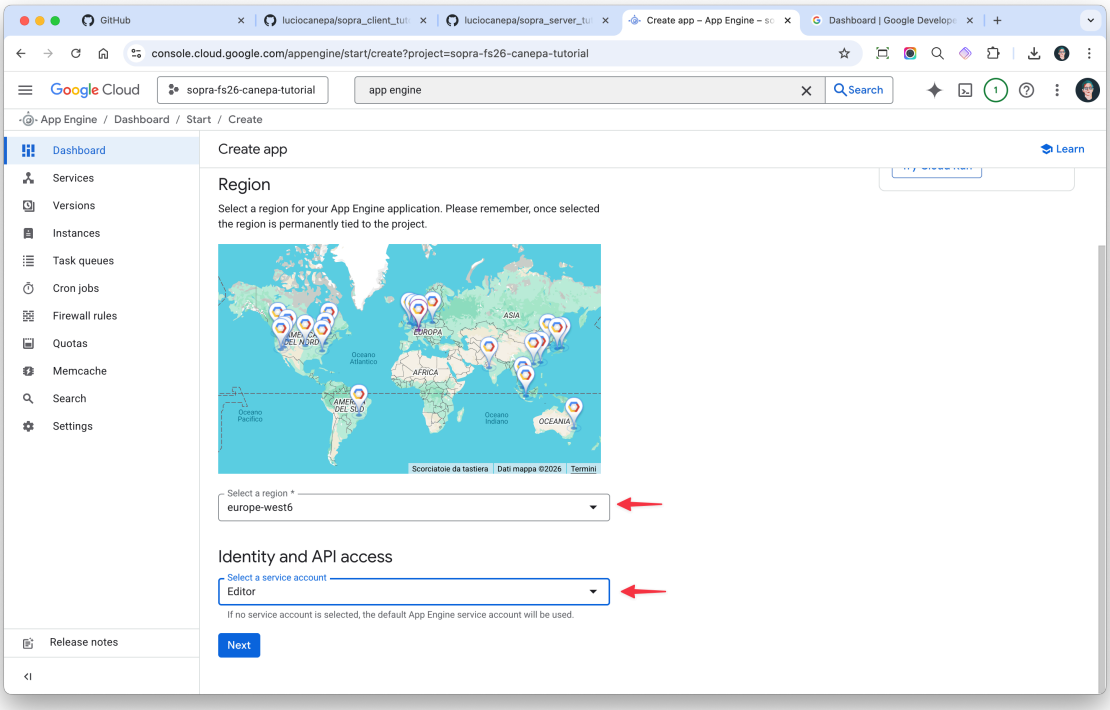
Use top bar to search for `App Engine`



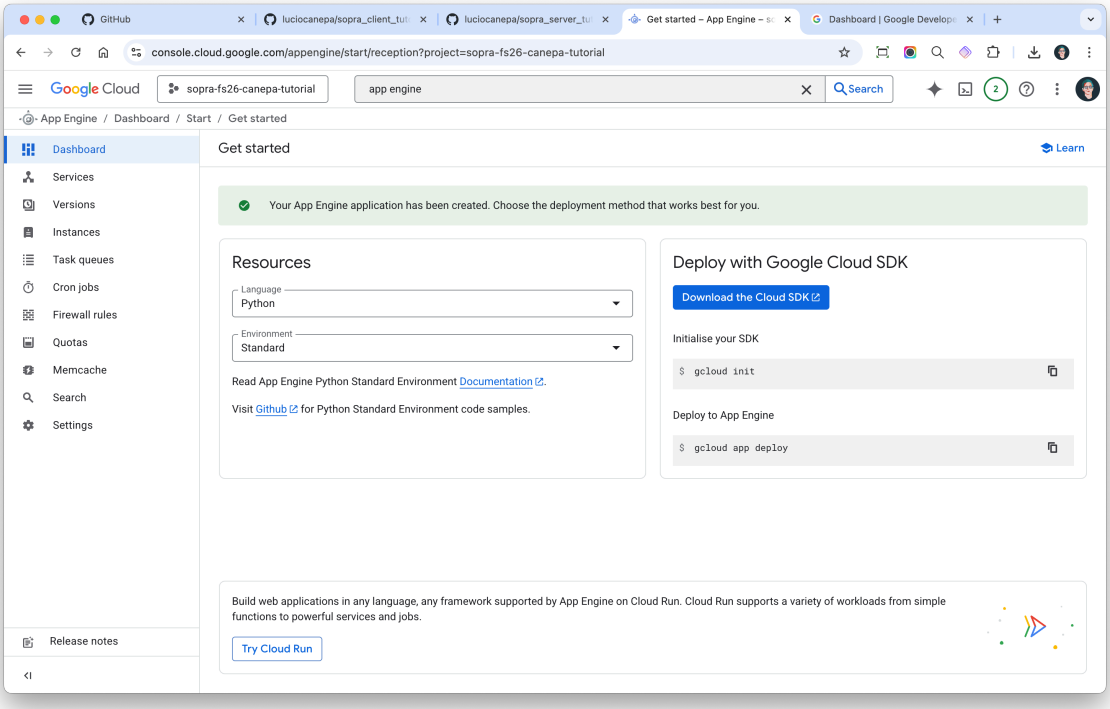
Create new application



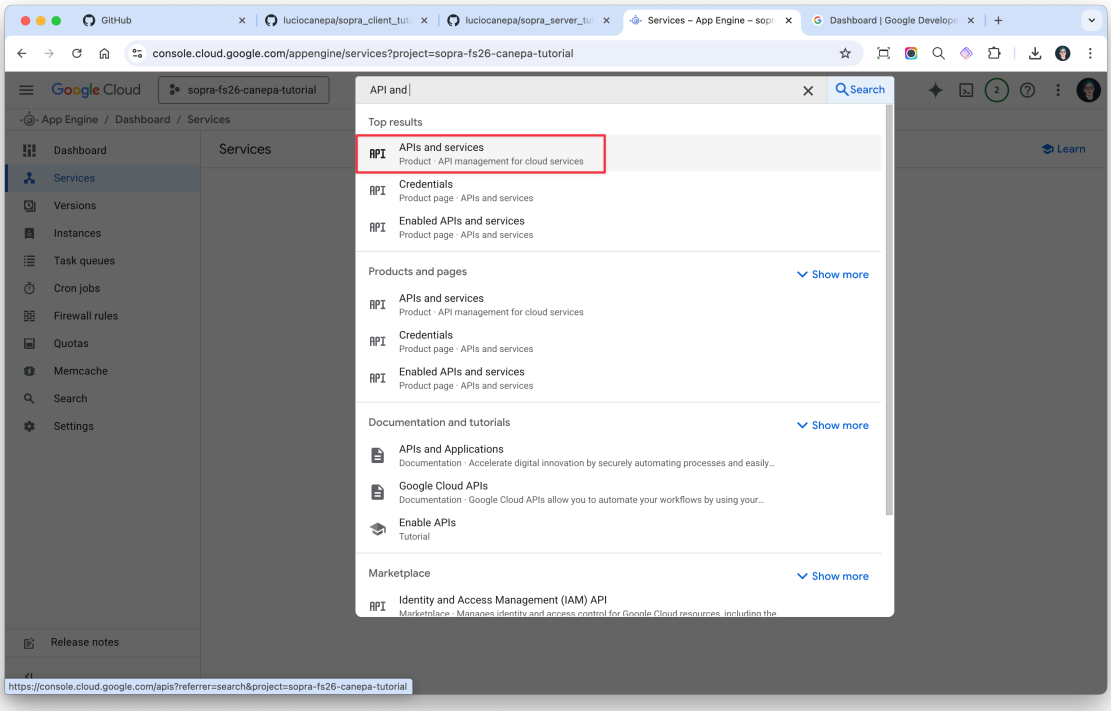
choose time zone `europe-west6` and connect the editor service account created before



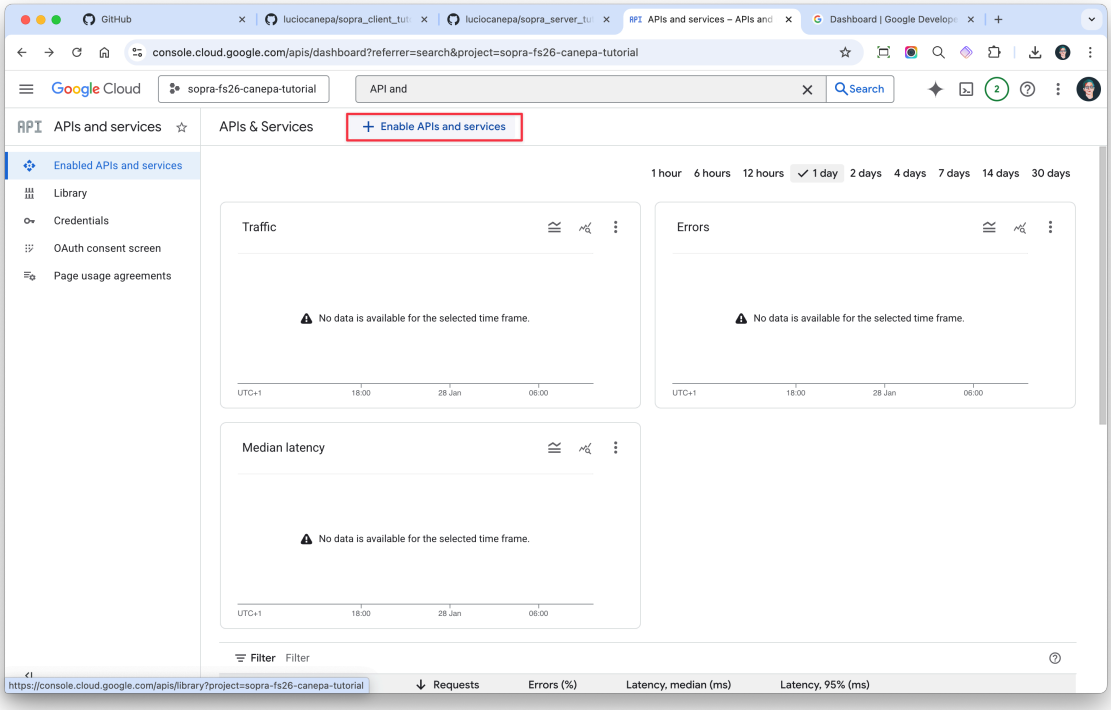
Confirmation screen



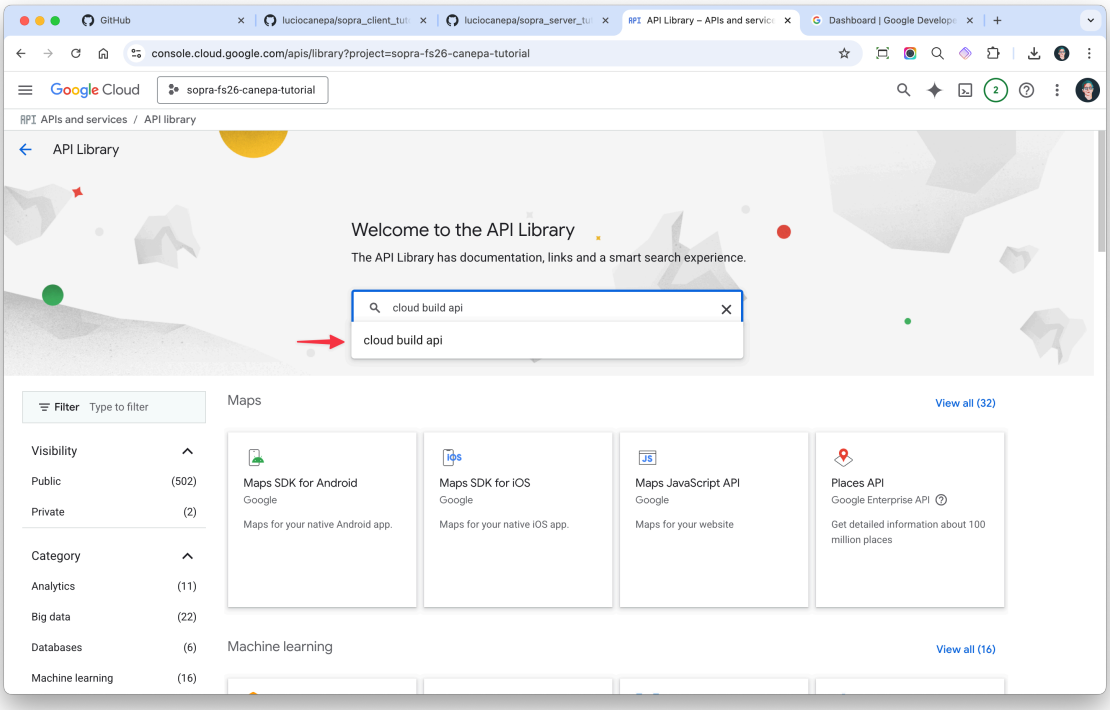
Use the top bar to search for `APIs and services`



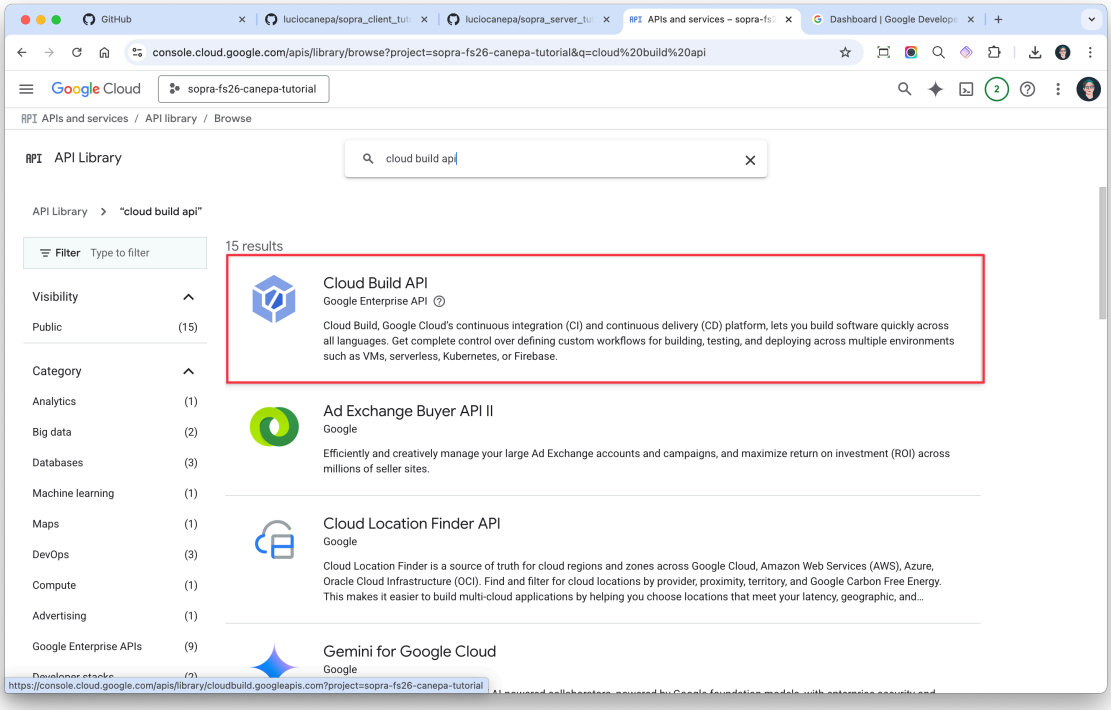
Click `+ Enable APIs and services`



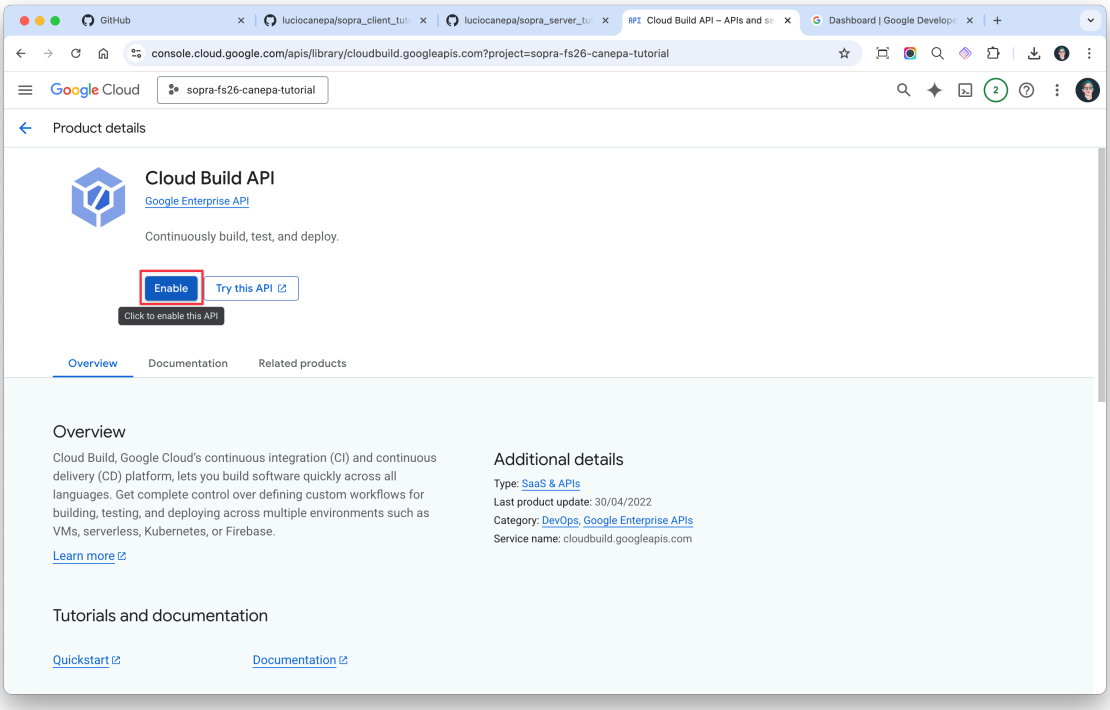
Search for `Cloud Build API`



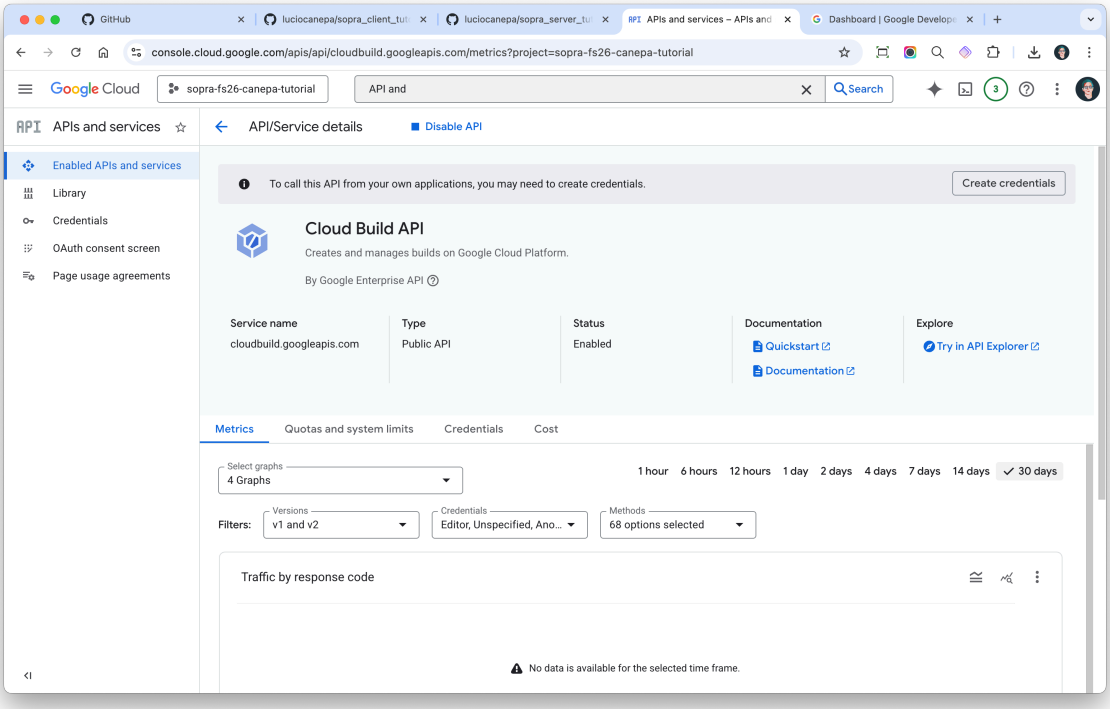
Click on `Cloud Build API`



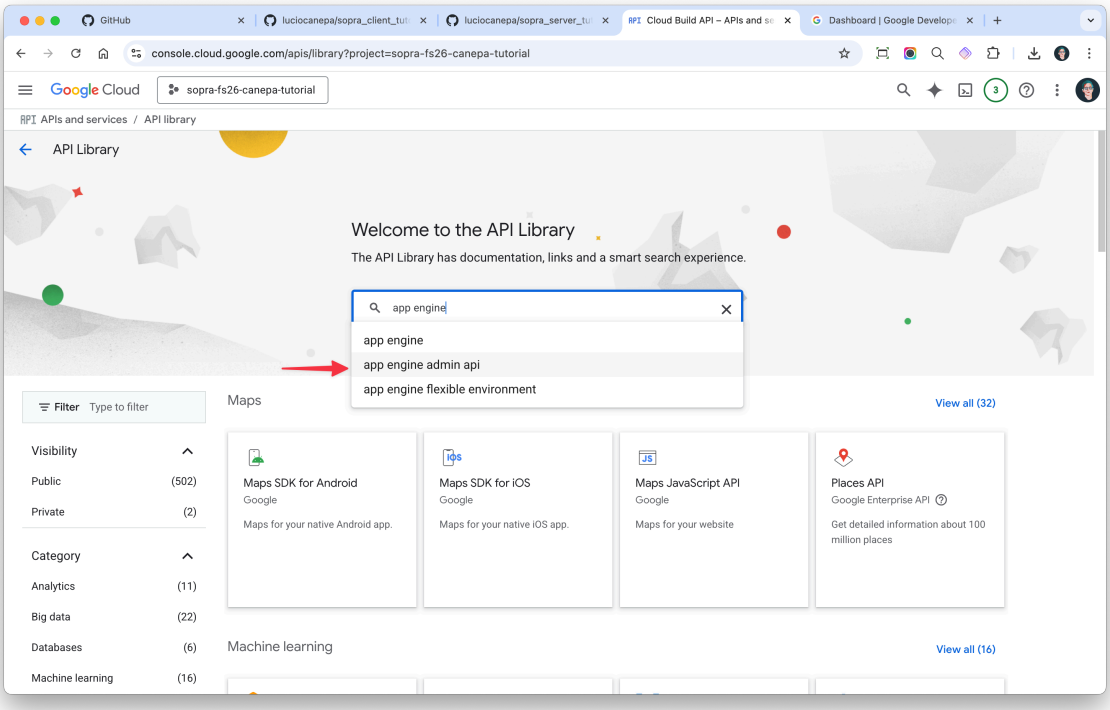
Enable it



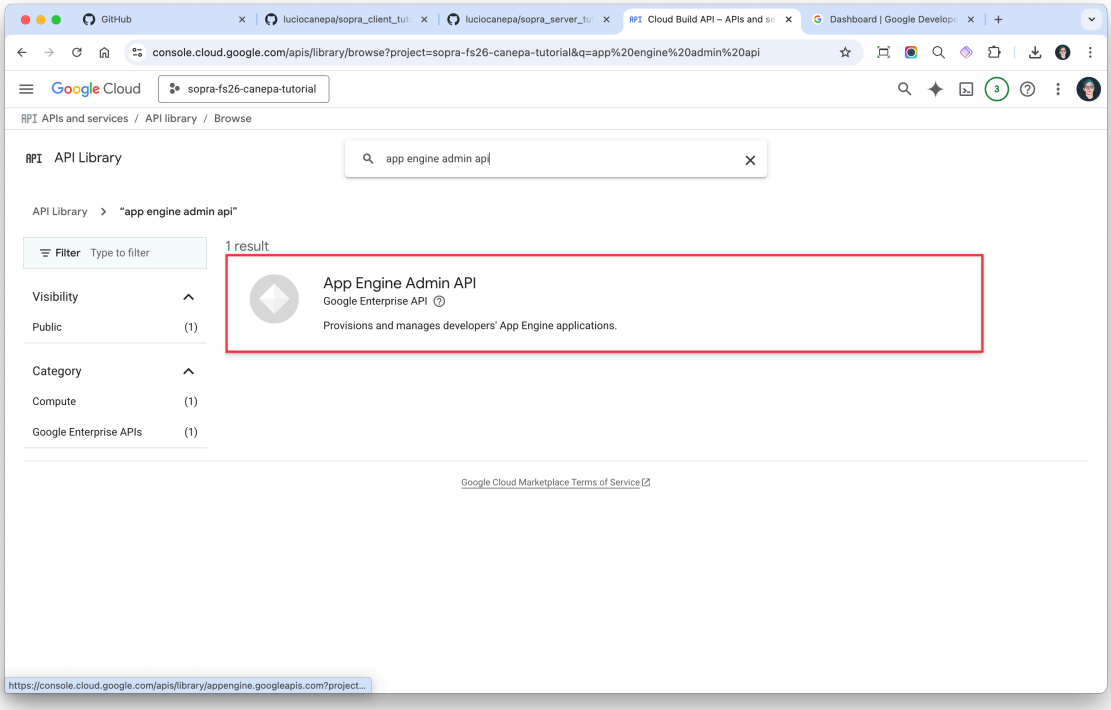
Confirmation screen: `Cloud Build API` is active



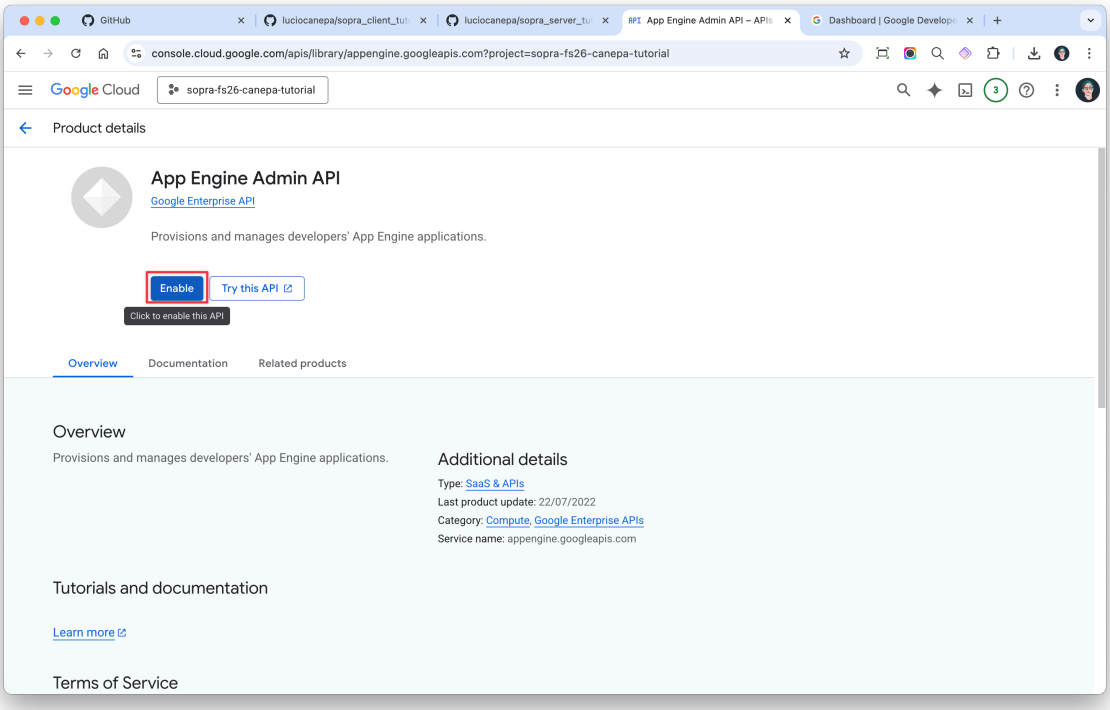
Back to the API library, search for `App Engine Admin API`



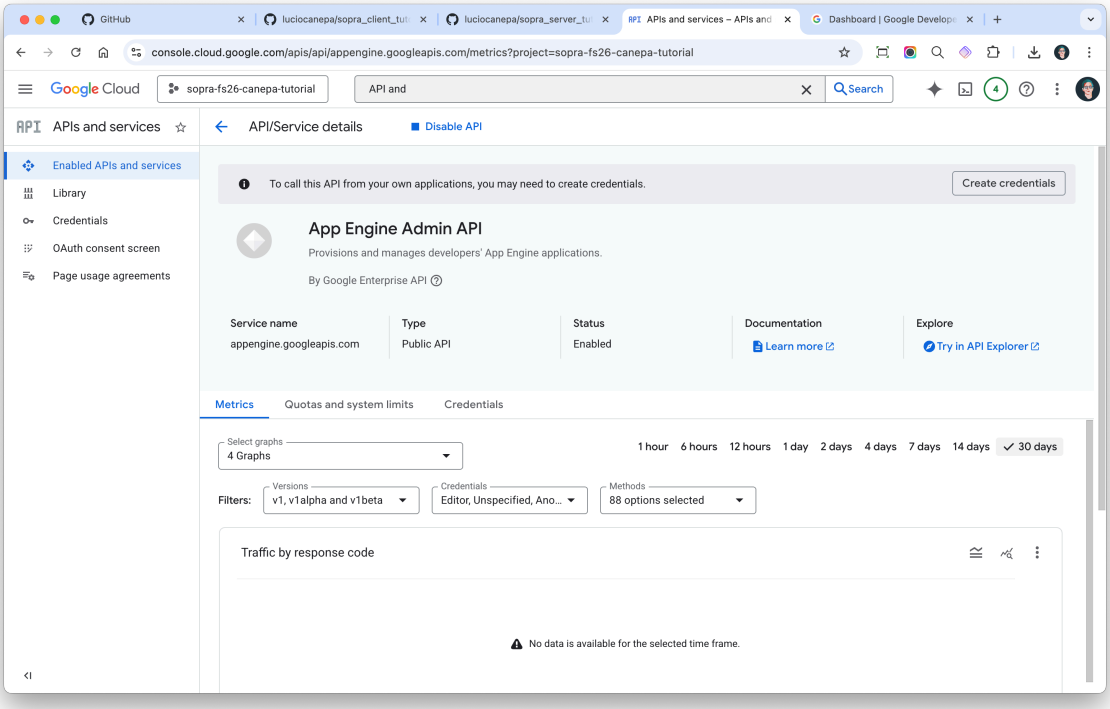
Click on `App Engine Admin API`



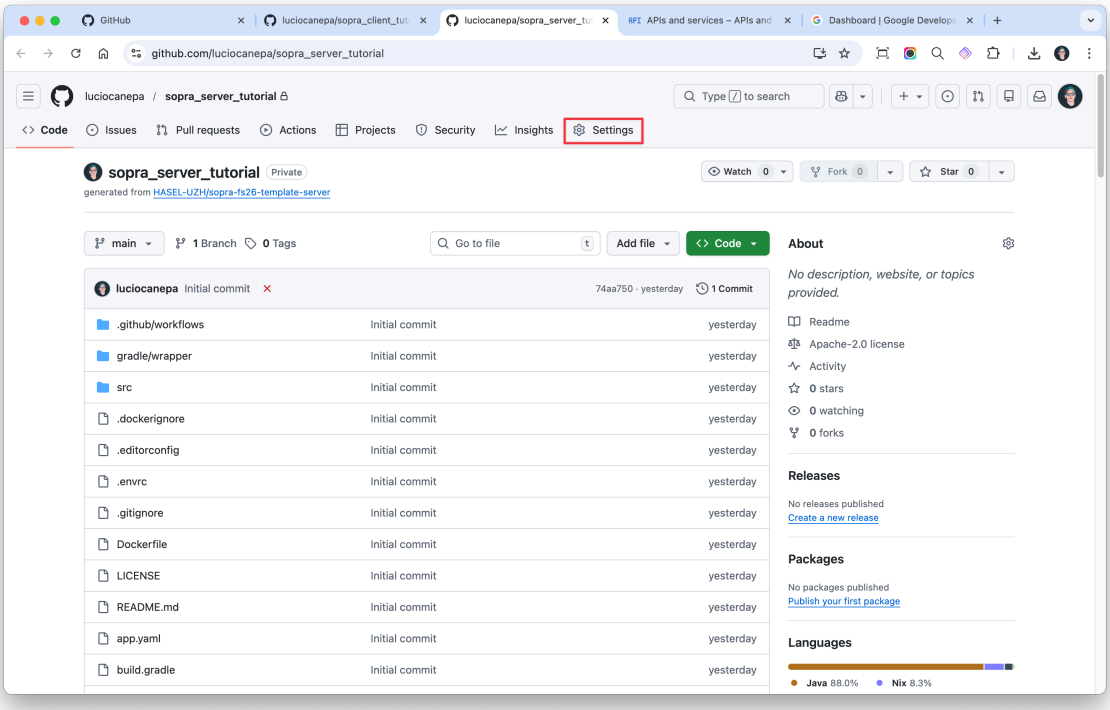
Enable it



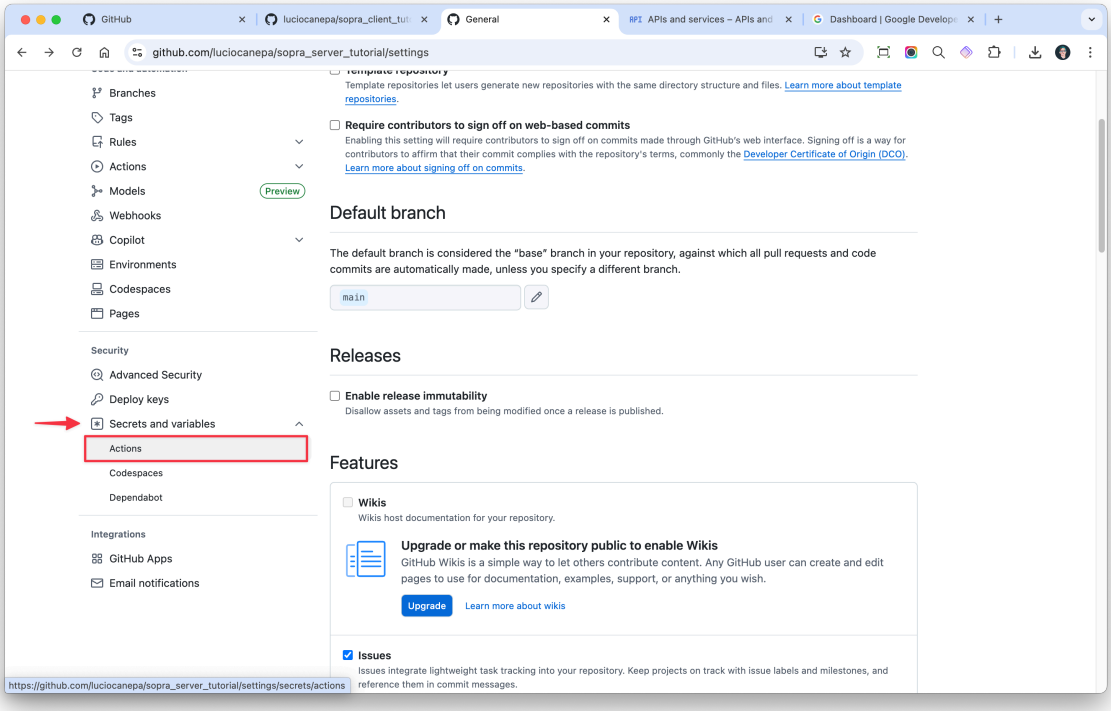
Confirmation screen: `App Engine Admin API` is active



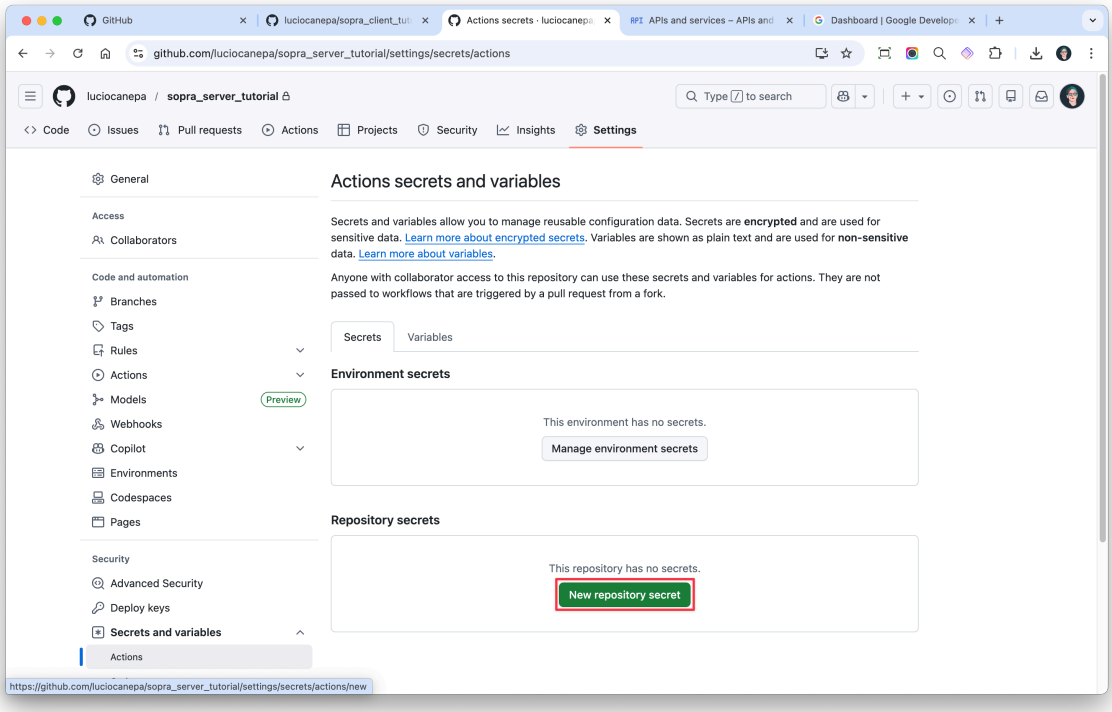
Go to the server repository on Github & go to the `Settings`



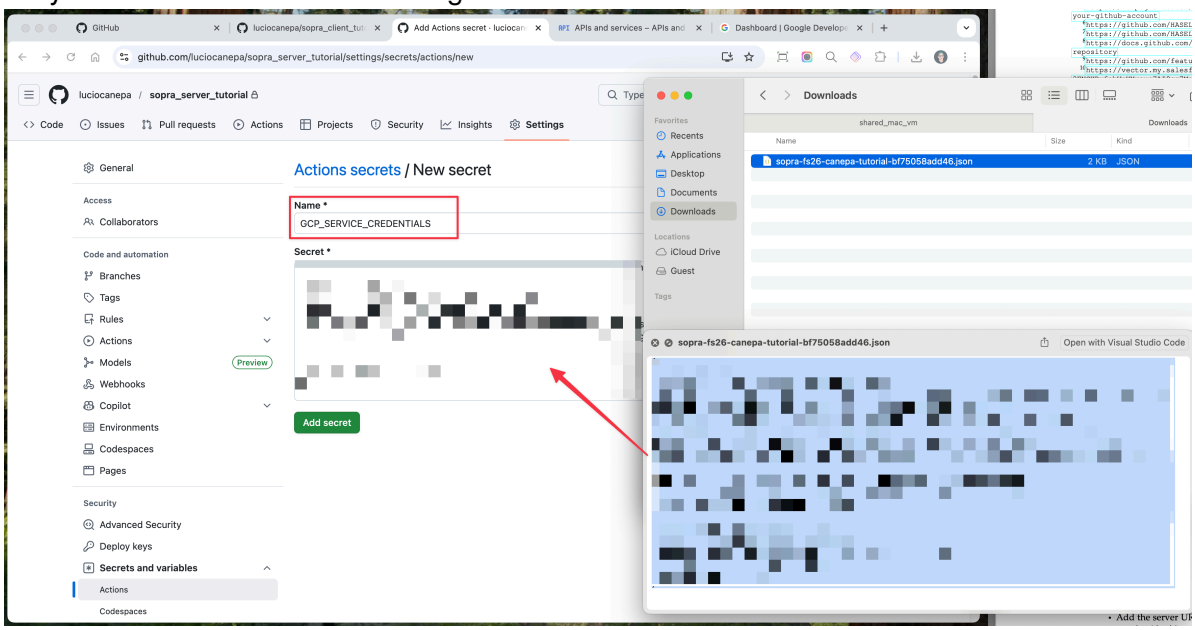
Look for `Secrets and variables` > `Actions`



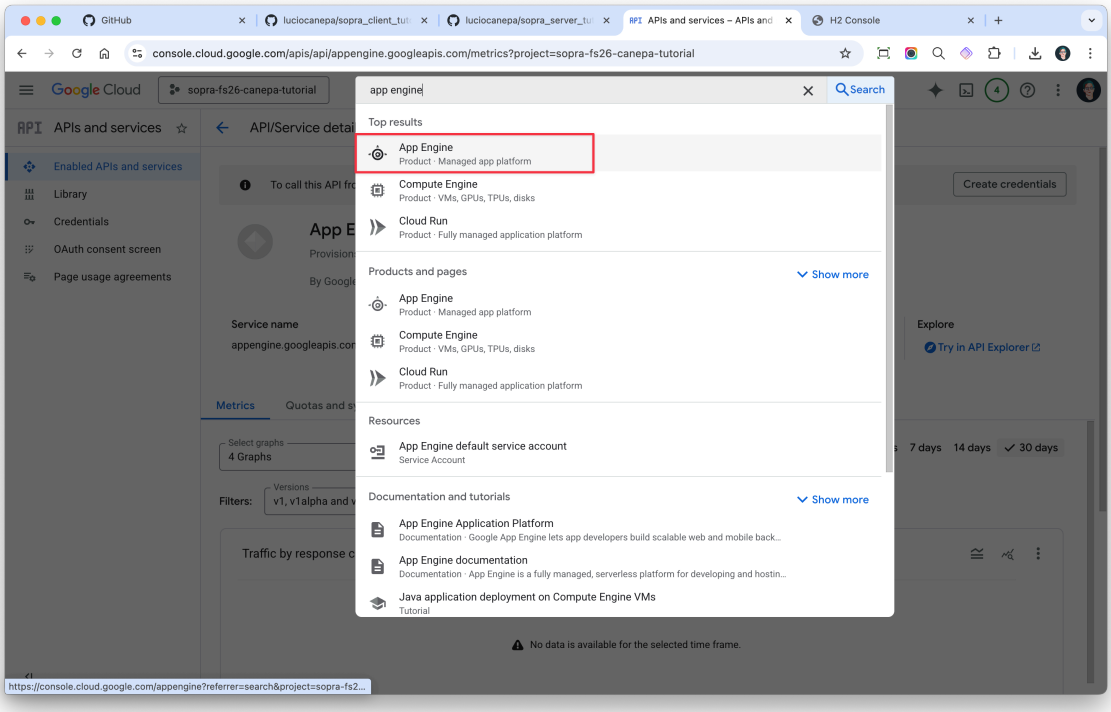
Ass a `New repository secret`



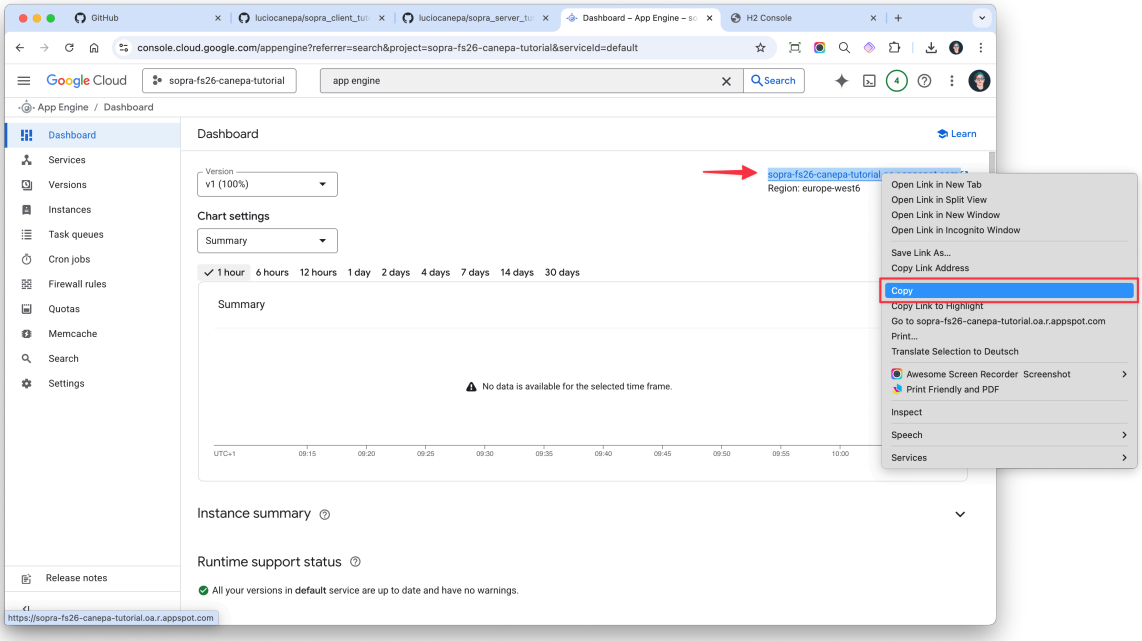
Name it `GCP_SERVICE_CREDENTIALS` and paste as value the content of the JSON file you download before from Google Cloud console



Back to Google Cloud console - search for `App Engine`



Here you find the link where your backend (server) will be deployed - copy it



Push to deploy & verify changes

Google Cloud console is now setup and the secrets are loaded in Github: the next time you

push to the remote repository, the server application will be automatically deployed (based on the `.gtihub/workflows`).
We can test this out by making some changes locally:

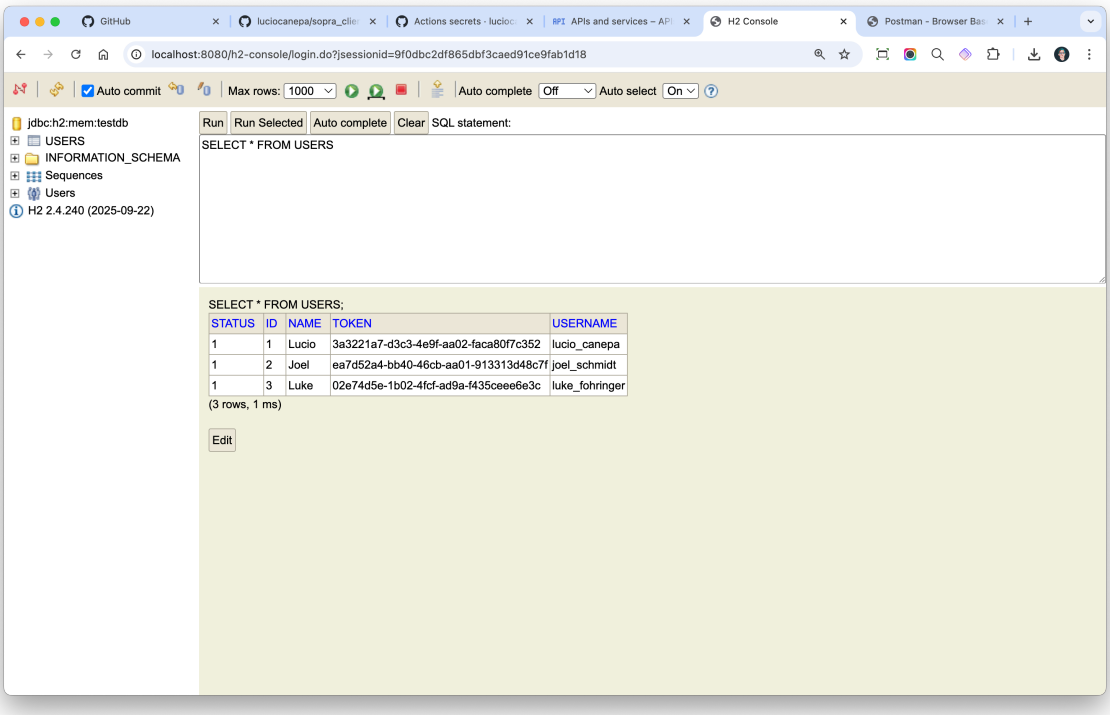
Local changes: add `GET /users/{id}` endpoint

---

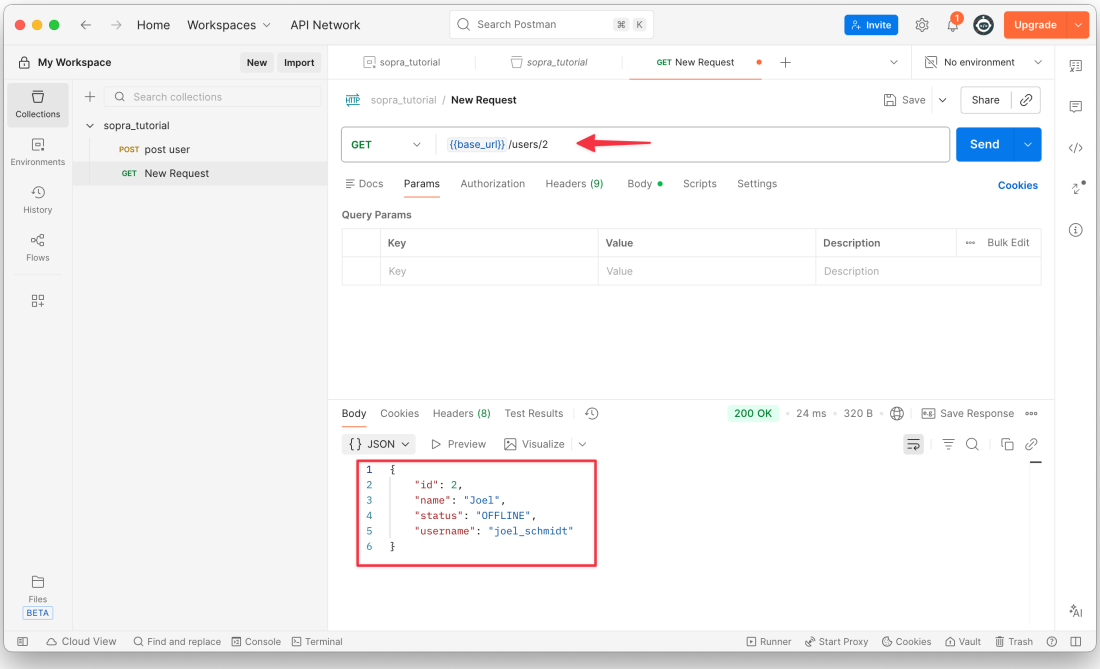
new endpoint in `users.controller` (I have also modified the correpsonding DTO and service files)

```java
@GetMapping("/users/{id}")
@ResponseStatus(HttpStatus.OK)
@ResponseBody
public UserGetDTO getUserById(@PathVariable("id") Long id) {
    User user = userService.getUser(id);
    return DTOMapper.INSTANCE.convertEntityToUserGetDTO(user);
}
```

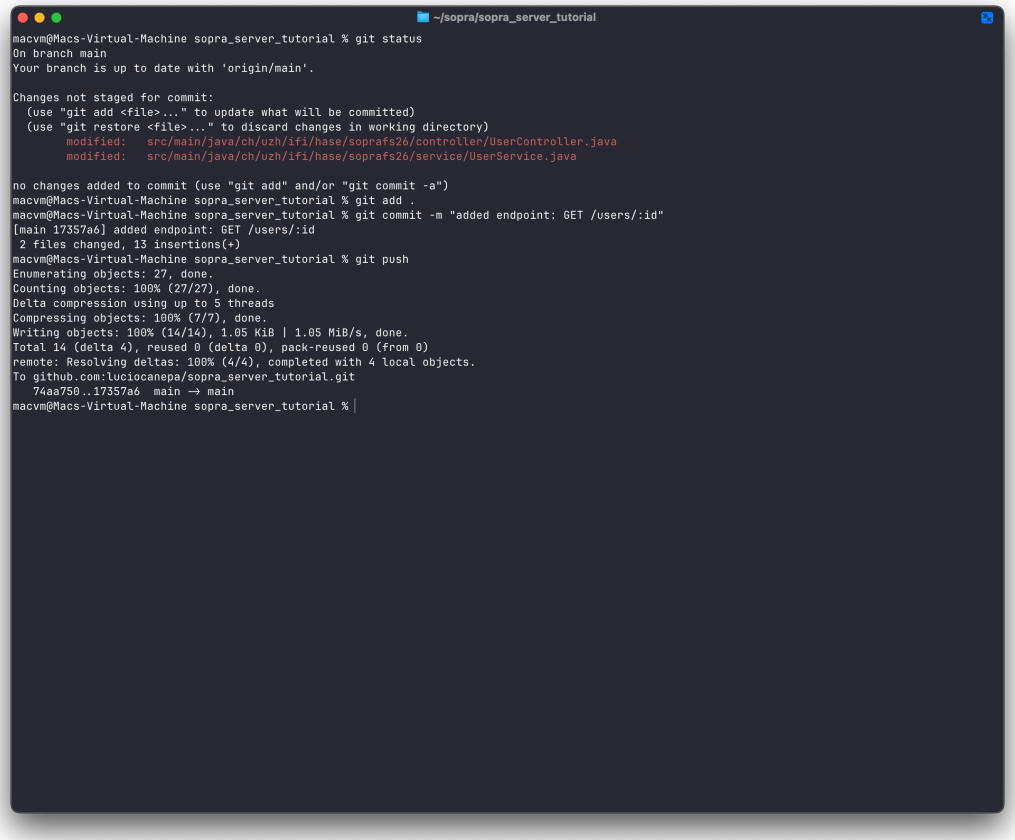
After creating 3 users, I can see the database locally



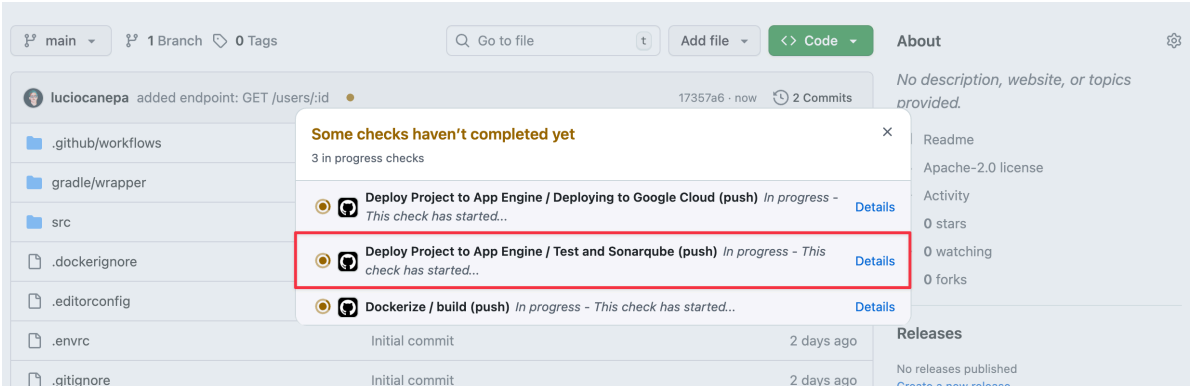
Test locally my new endpoint



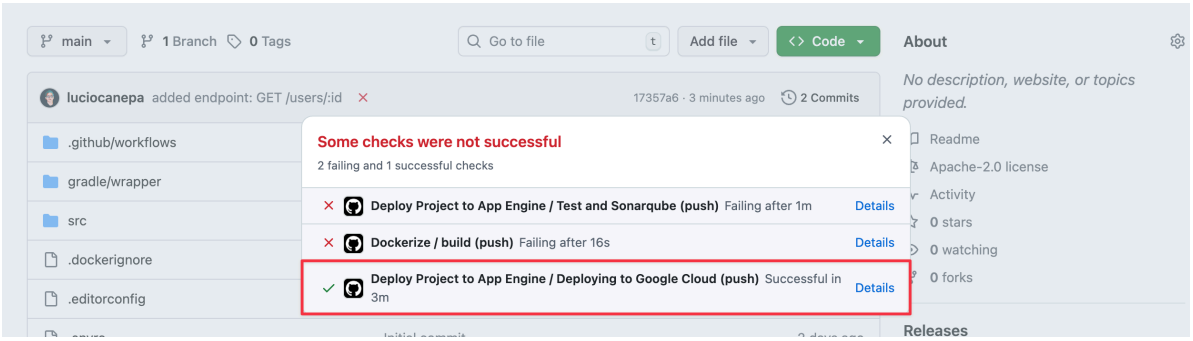
Commit and push the changes



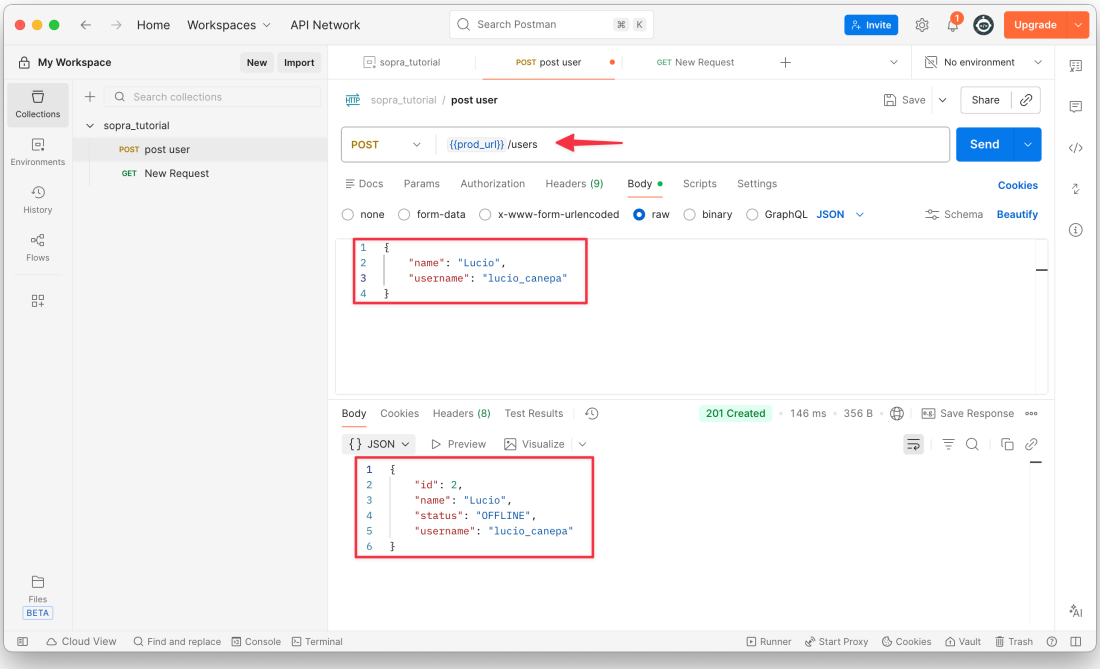
We are interested in this deploy workflow



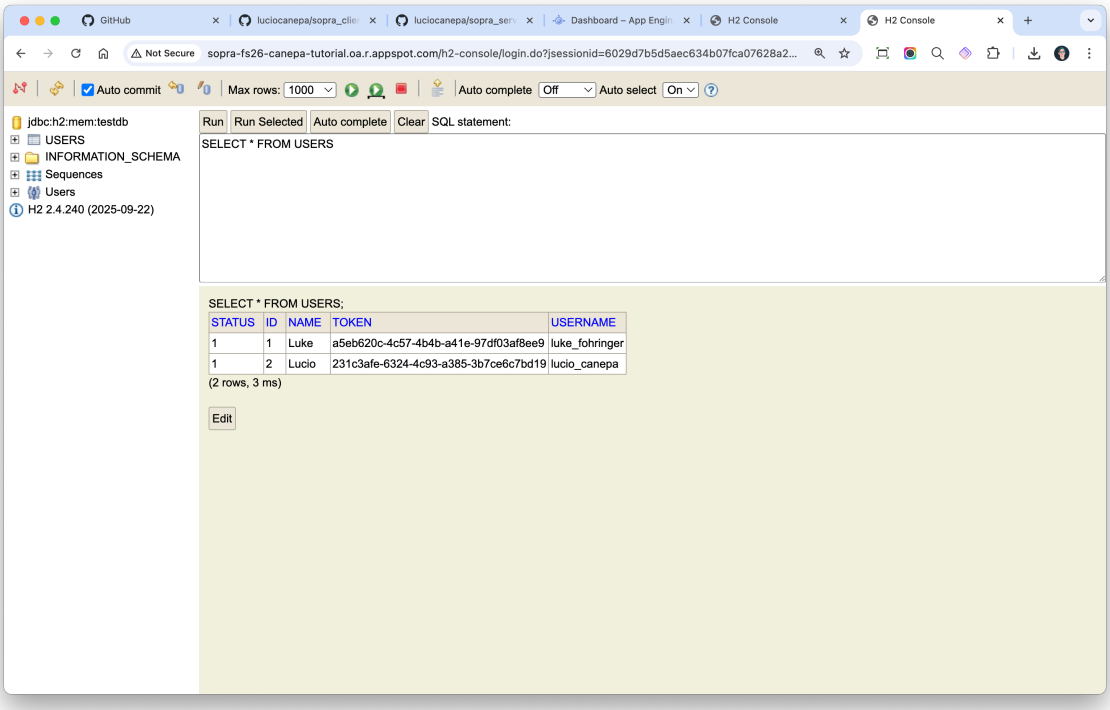
Confirmation: the `Deploy to App Engine workflow succeded` - for now ignore the other 2



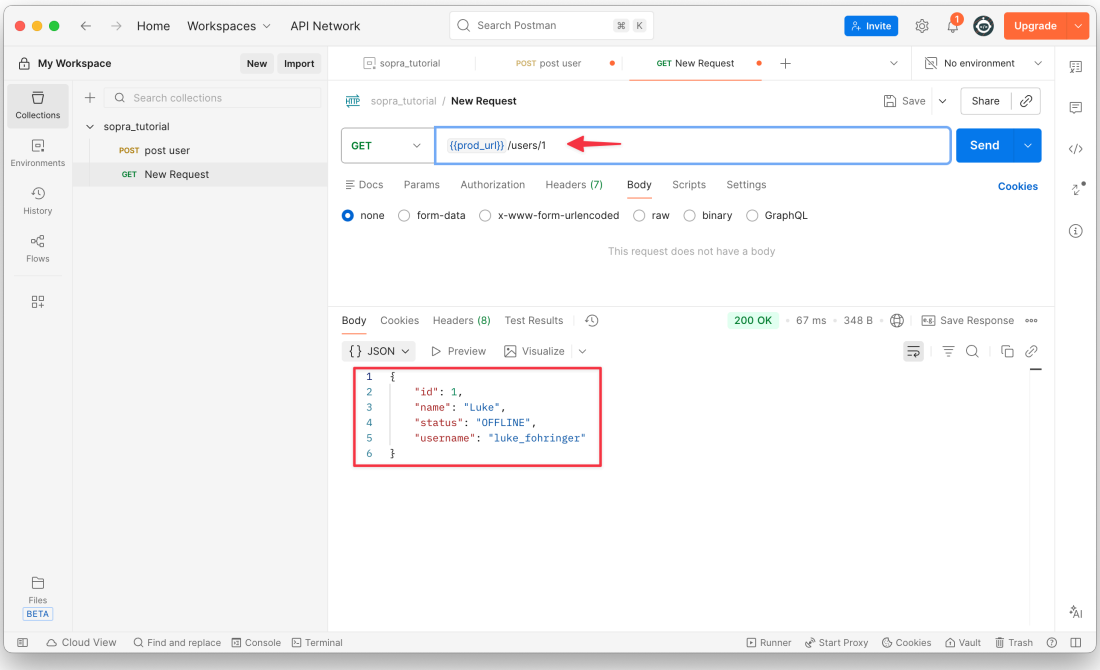
Now we can test the new endpoint in production (prod_url is the link we copied before from `App Engine`) I create 2 users (here you see how to POST a new user)



from `prod_url/h2-console`, you can see the tables as you did locally (to access use the same authentication)



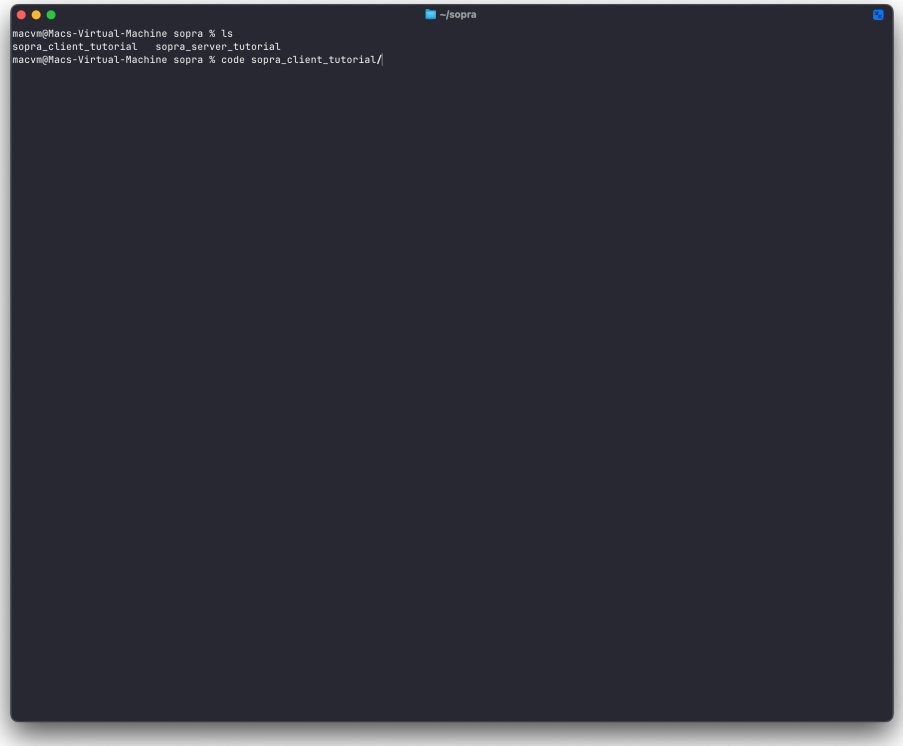
Test the new endpoint in production



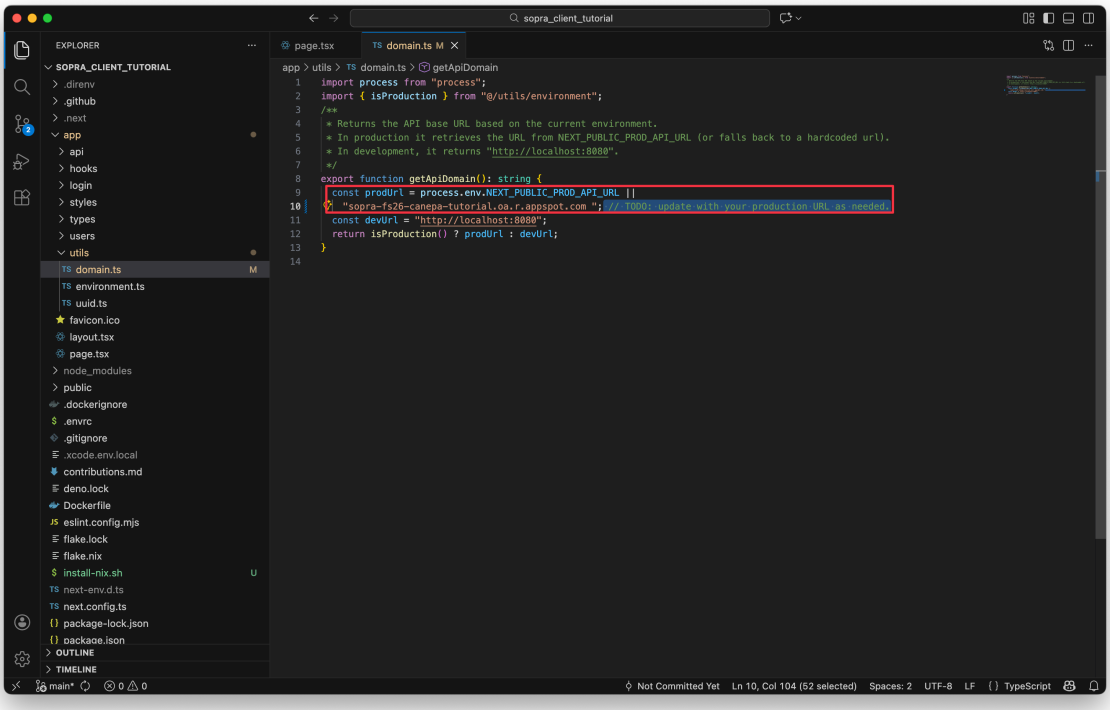
Update fallback prodUrl in the client repository

Make sure to have the server production URL copied to the clipboard (you find it at Google Cloud console > `App Engine`)

Open the client repository in an IDE of your choice



Look for `app/utils/domain.ts` and replace the hardcoded fallback value to the URL you just copied - you completed the TODO commented out there

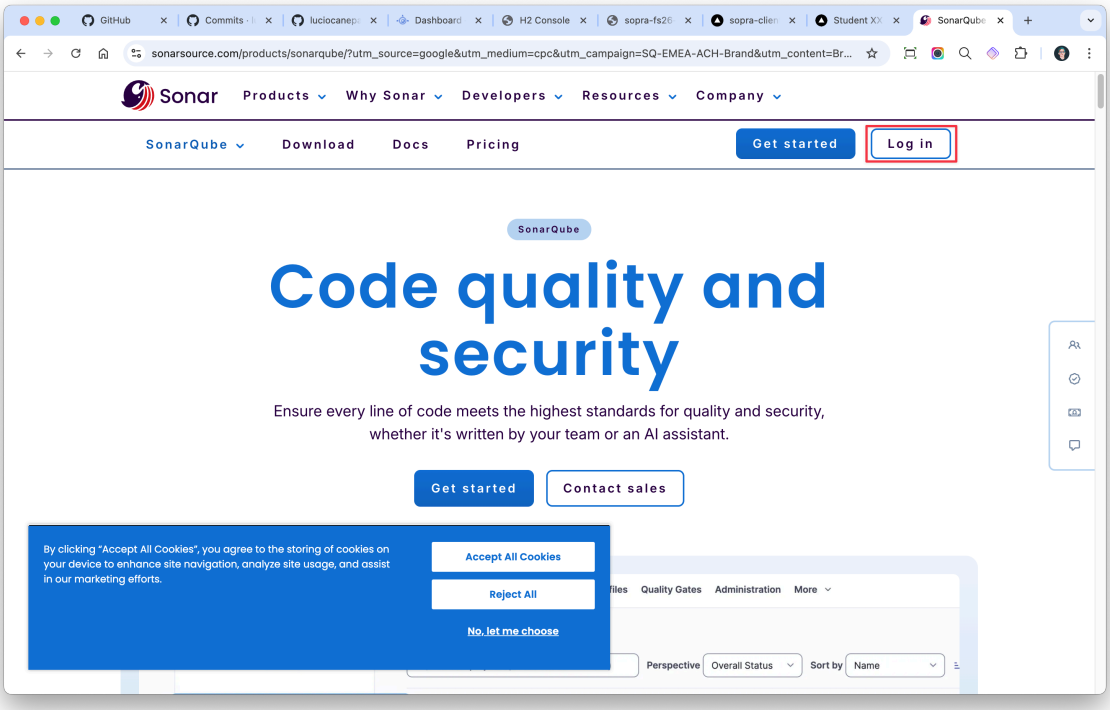


Ideally the server URL in production comes from NEXT_PUBLIC_PROD_URL that we are going to set-up later (in the dedicated Vercel part of this tutorial). This only serves as a fallback if, for some reason, the environment variable set in vercel is not retrieved correctly.

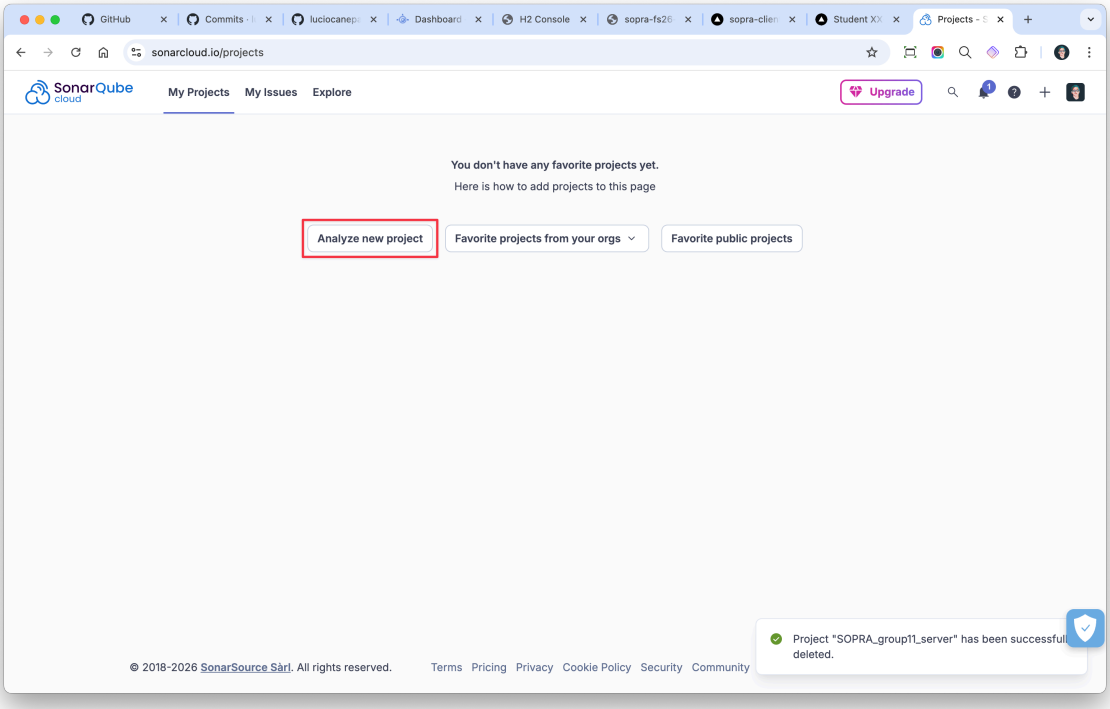
Set-up Sonar cloud and connect it to the sever repository (optional)

Go to https://www.sonarsource.com/

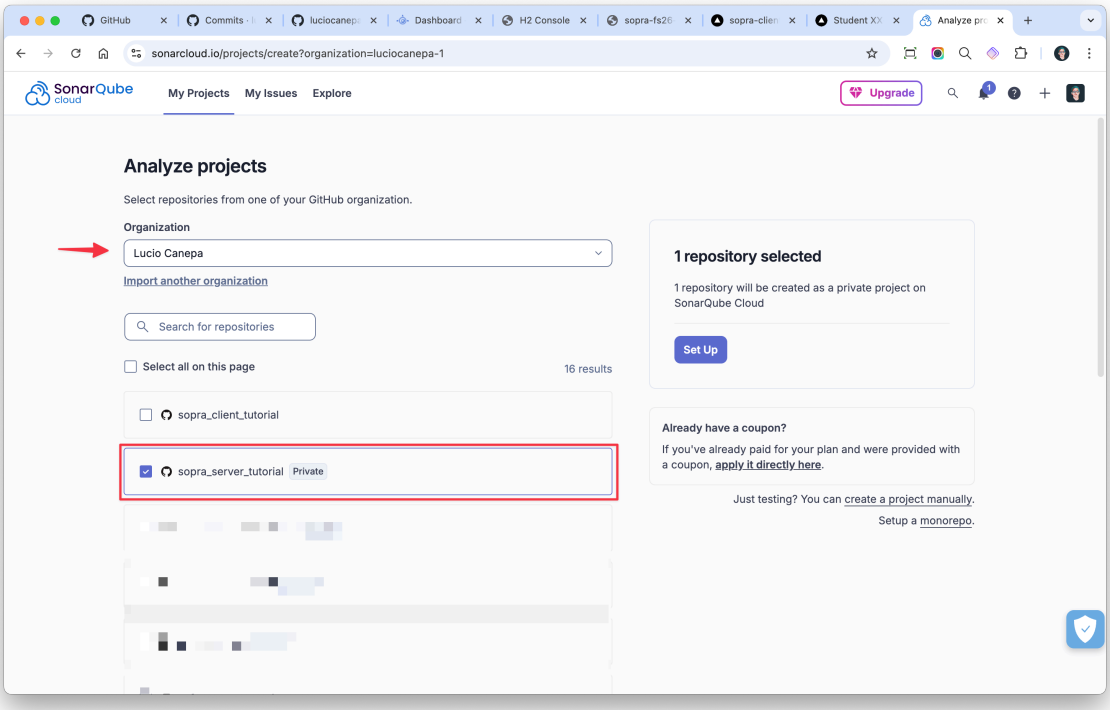
Register or login (best if with your Github account)



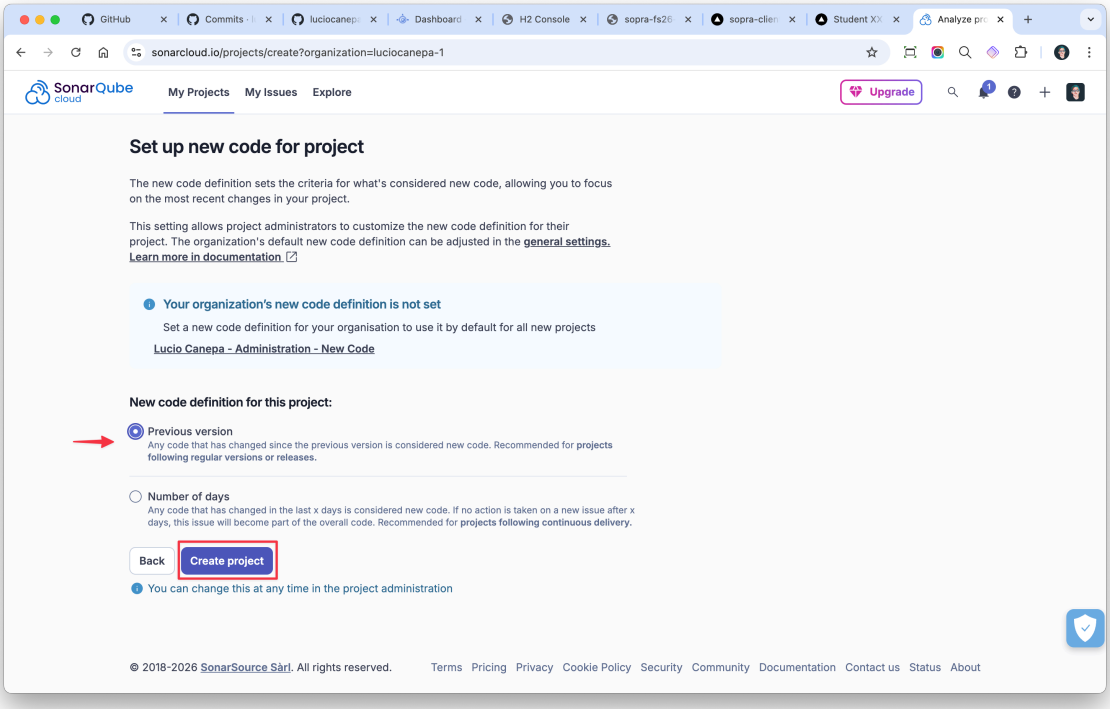
Click `Analyze new project`



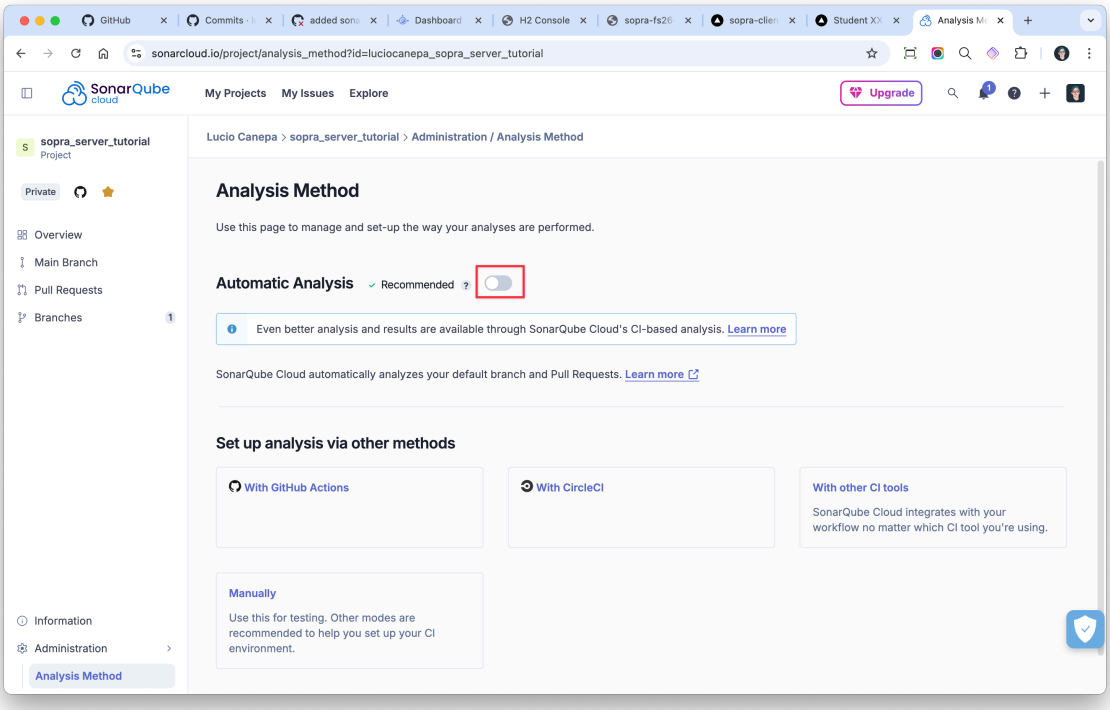
Choose your organization and link the server repository



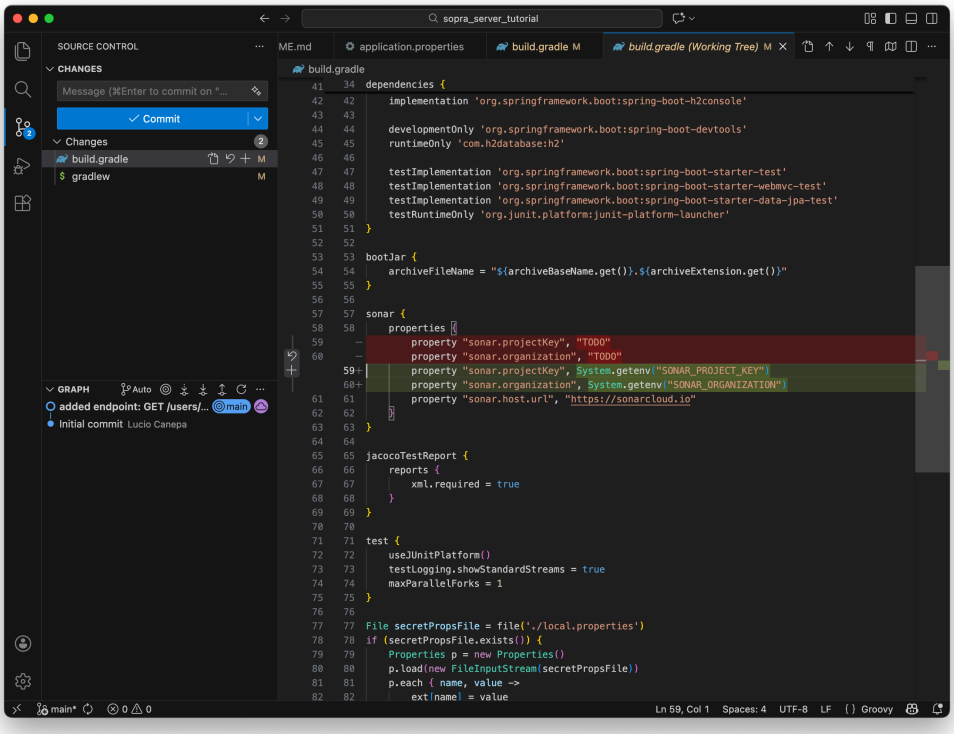
Leave default `Previous version` toggle & click `Create project`



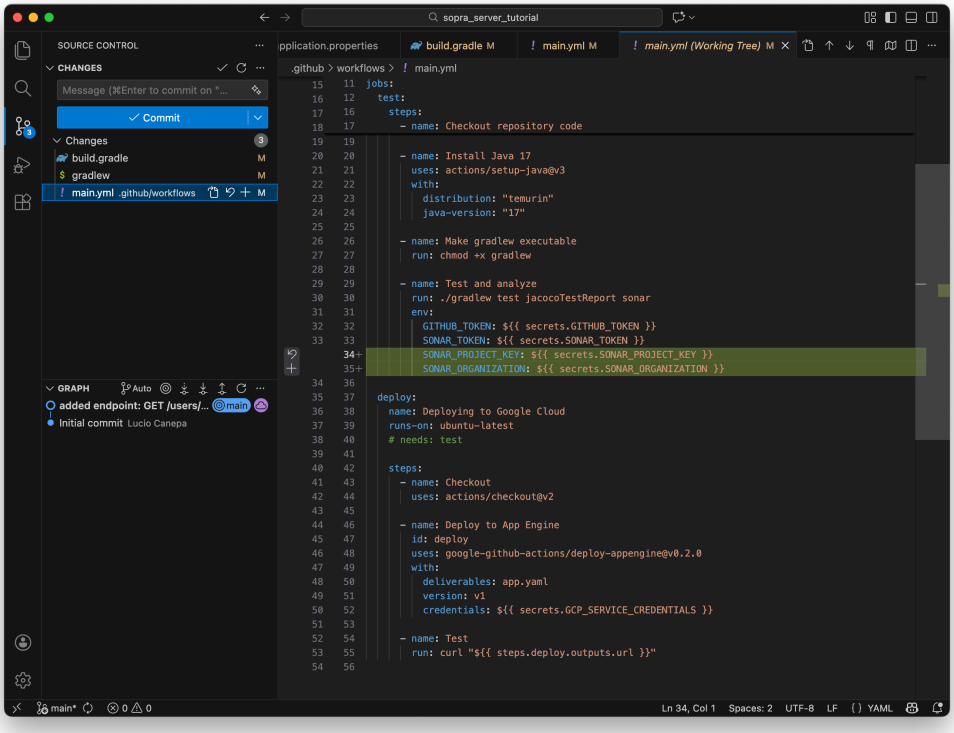
Under `Administration` > `Analysis Method` turn off the toggle `Automatic Analysis` (since we are going to push it via CI)



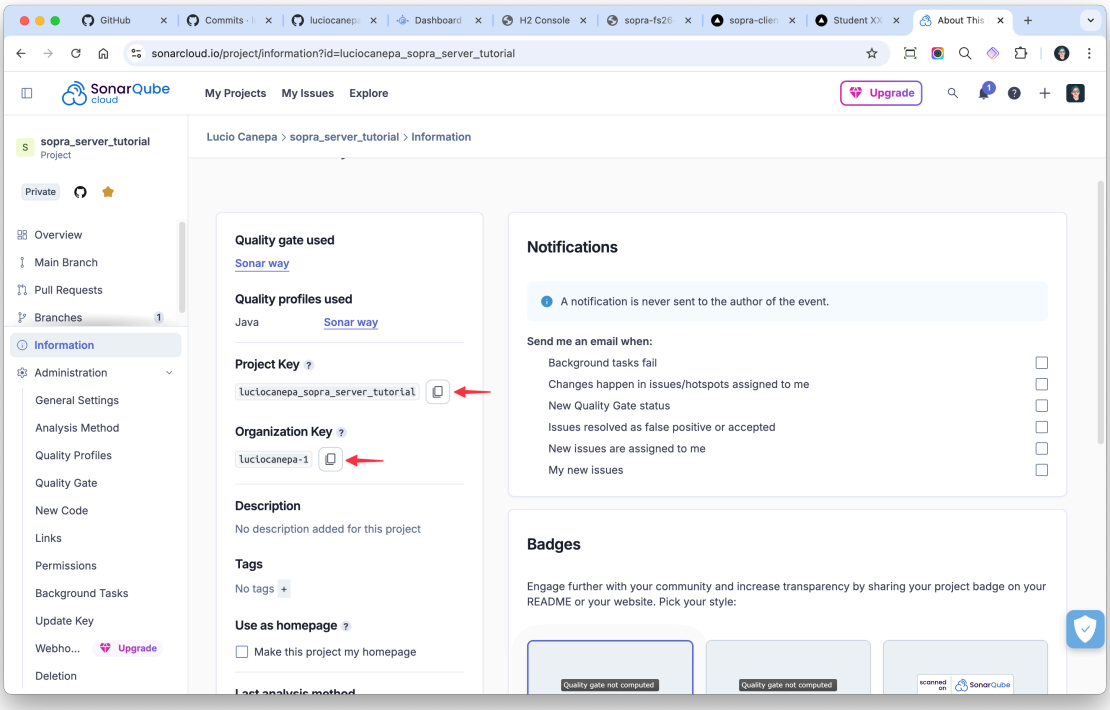
Modify your `build.gradle` as in the diff (server repository)



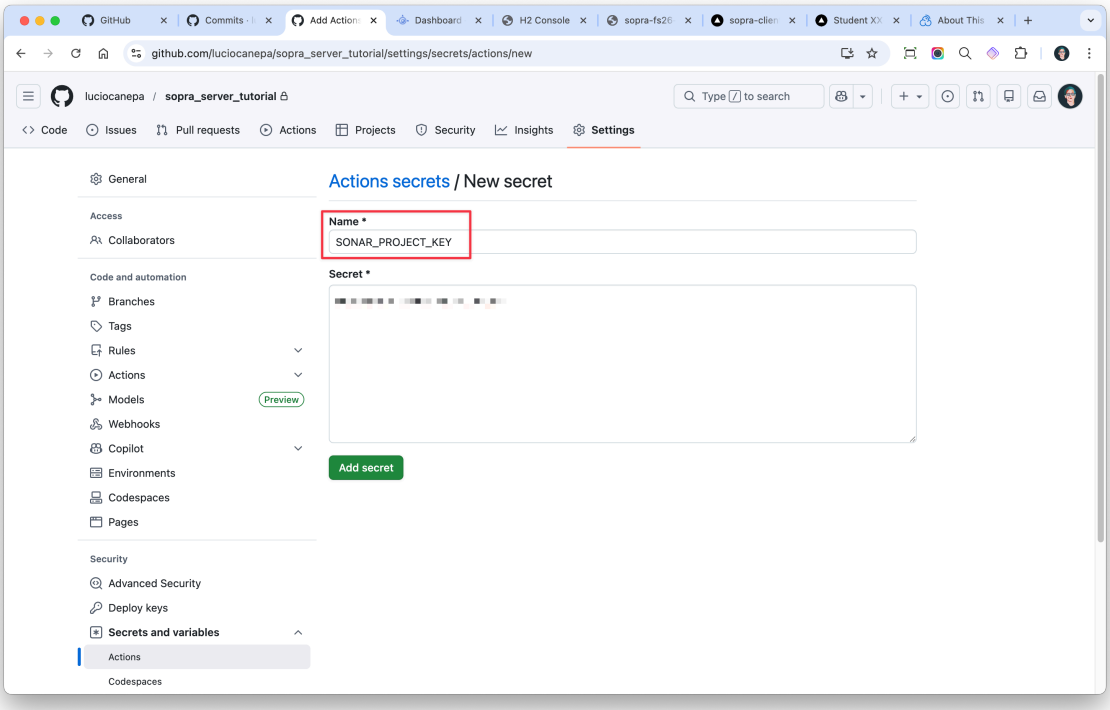
Modify your `main.yml` worflow file as in the diff (server repository)



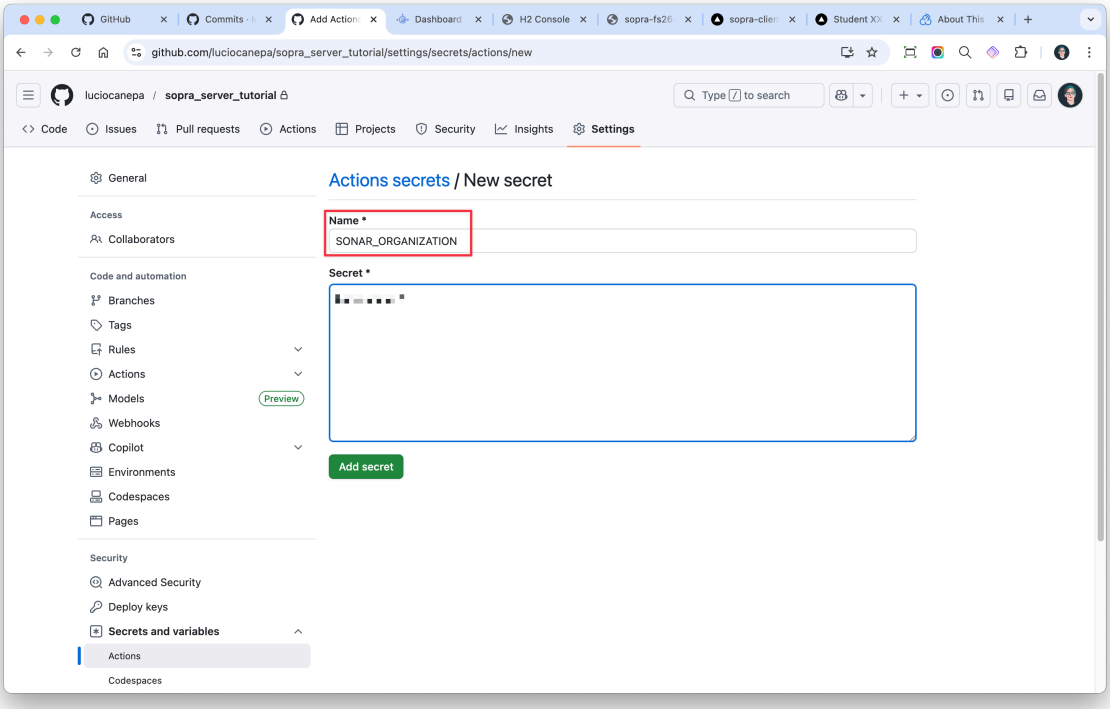
We now need to collect the secrets to connect the server repo to sonarcloud. You find the `Project Key` and `Organization Key` under `Information` - copy them



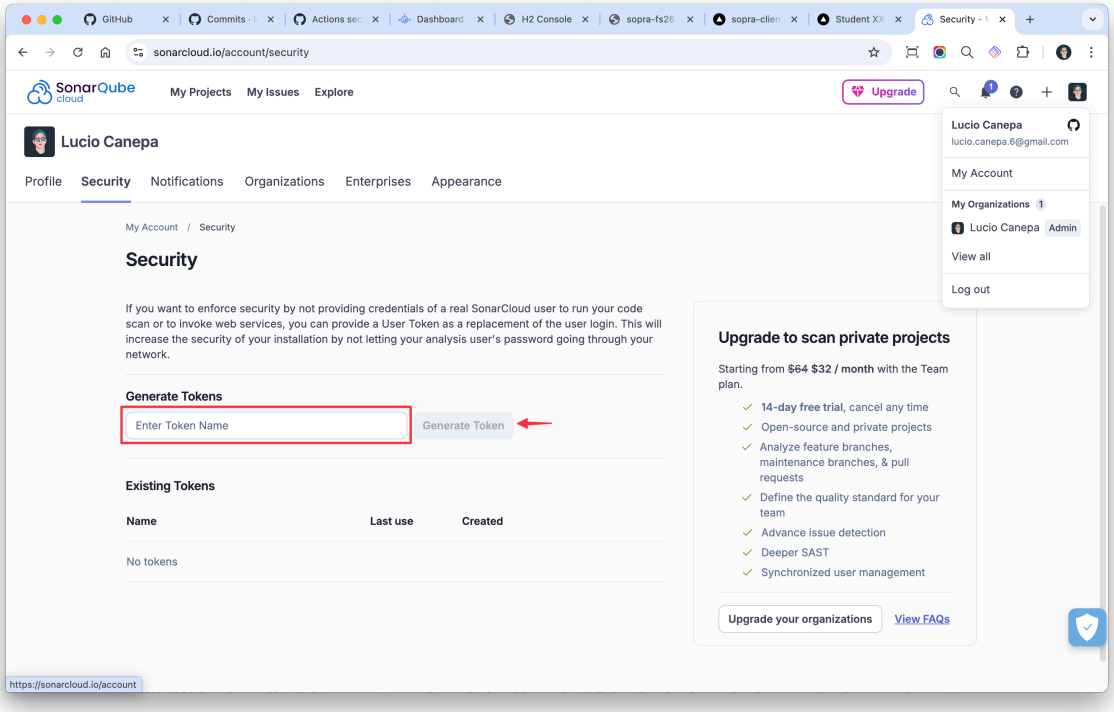
Add a new secret to the server repository: `SONAR_PROJECT_KEY` (`Project Key`)



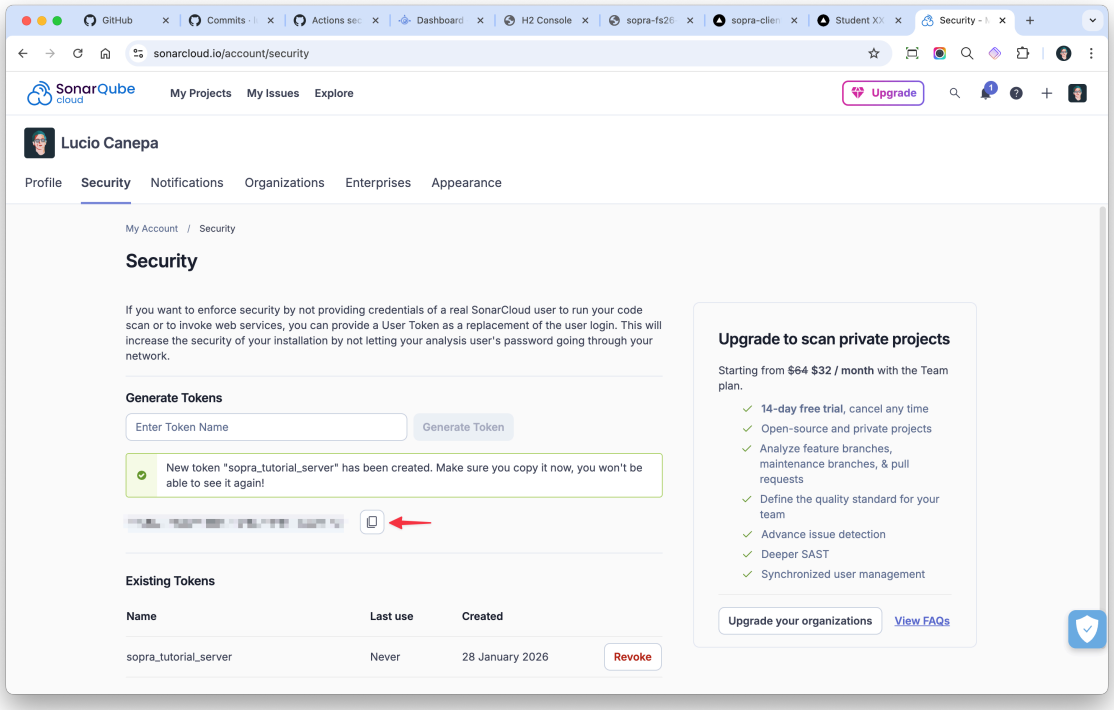
Add new secret: `SONAR_ORGANIZAZION` (`Organization Key`)



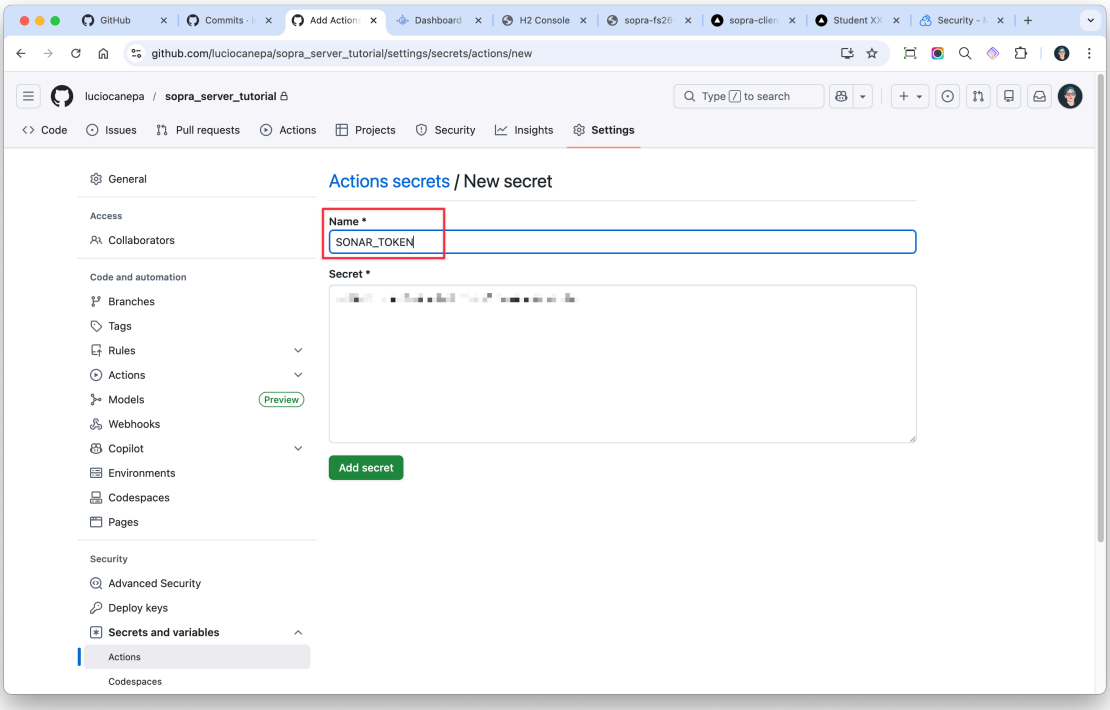
To get the token, click on your profile picture > `My Account` > `Security` - enter a token name (whatever you prefer) > click `Generate Token`



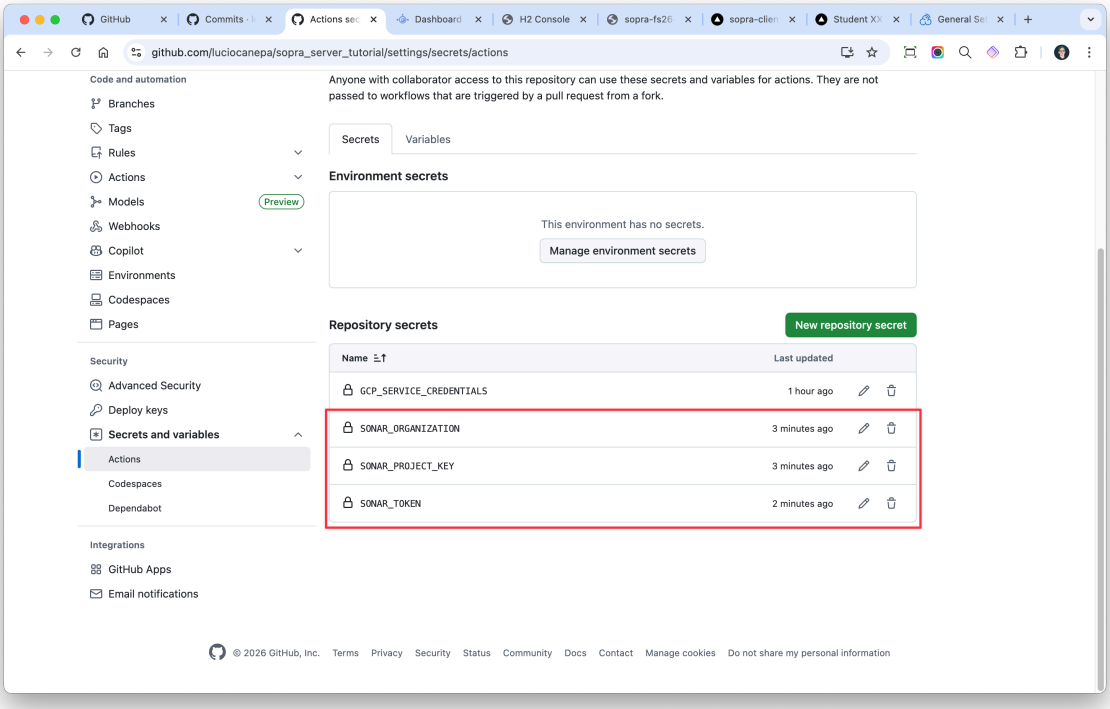
Copy the generated token



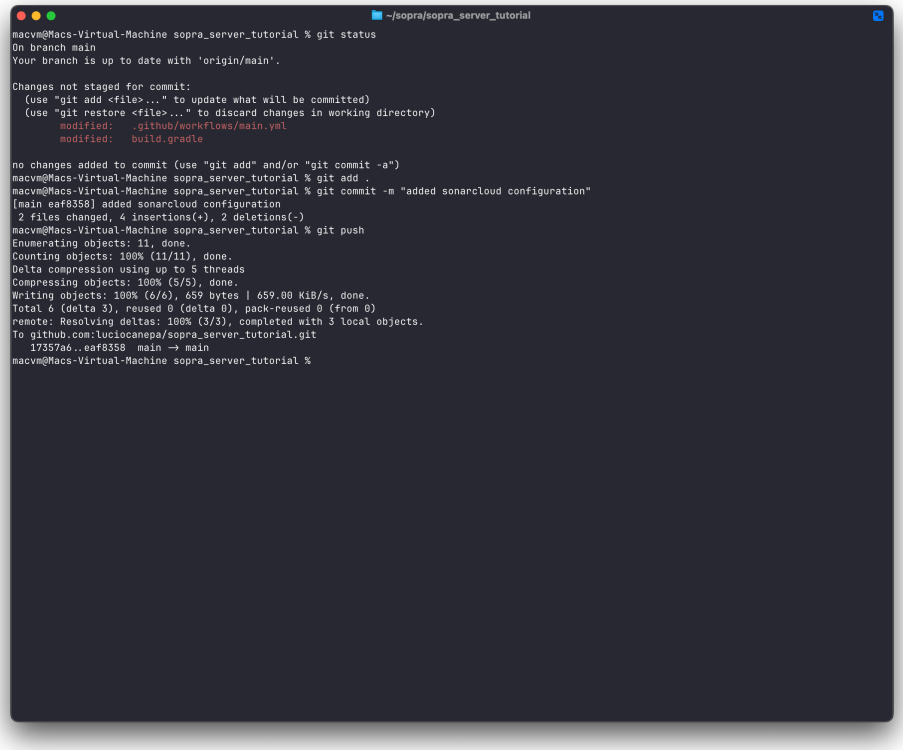
Add new secret: `SONAR_TOKEN`



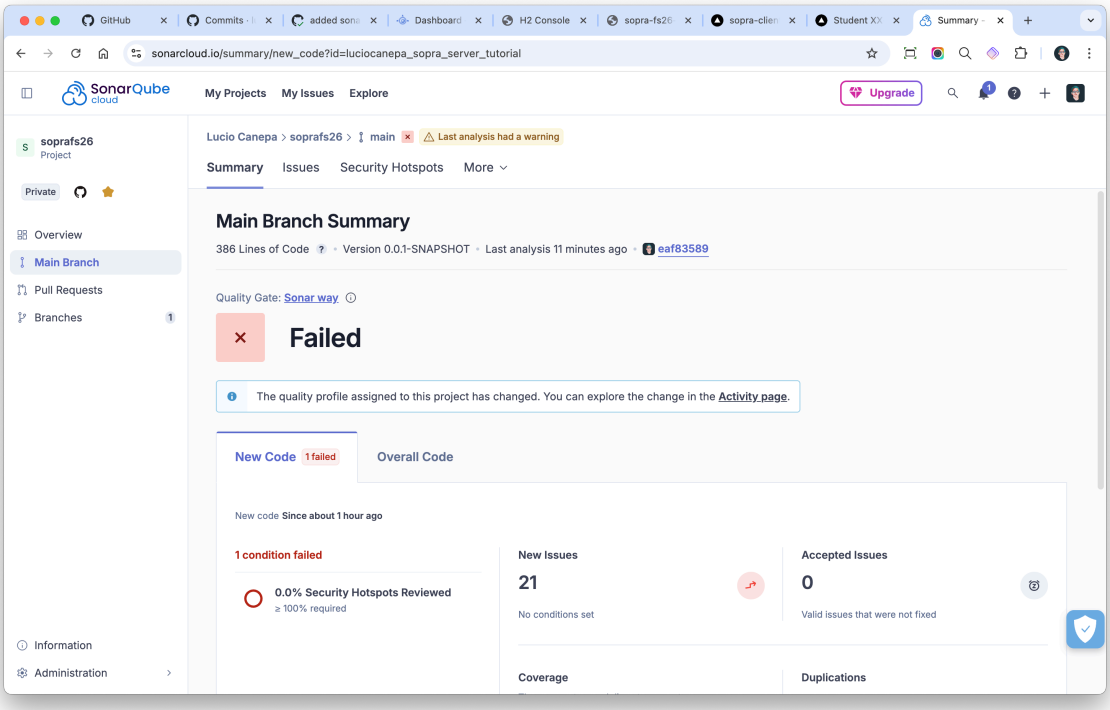
At this point you should have added 3 new secrets to your server repository



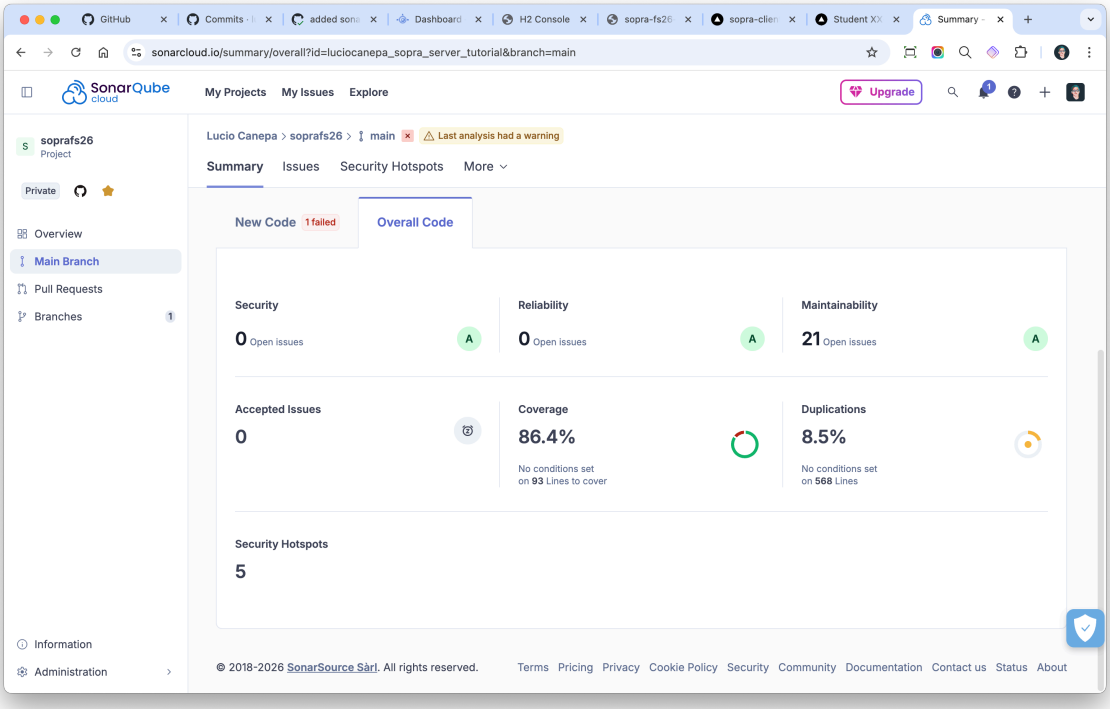
Commit and push the changes to your server repository. Now both the Google cloud deploy and the `Test and Sonarqube` should be successfull



In SonarQube, under your project, you can see the results



and navigate between them



5.2 Set-up Vercel and link it to the client repository

To set up the frontend on Vercel, the following components must be configured (as in the step-by-step tutorial):

Platform & Project

- Vercel Project: Linked directly to your GitHub client repository.
- Deployment URL: The live production link for the web application.

GitHub Secrets (for CI/CD)

- `VERCEL_TOKEN`: Authorized access token for deployment.
- `VERCEL_ORG_ID`: Your Vercel account/organization identifier.
- `VERCEL_PROJECT_ID`: The specific ID for the client project.

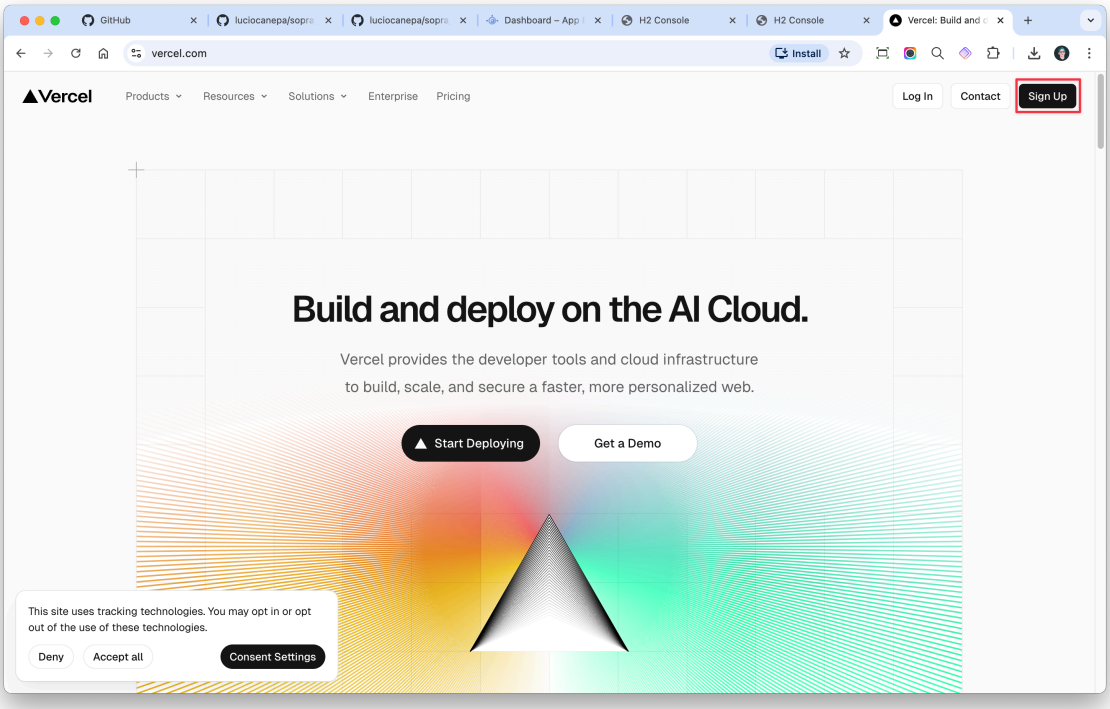
Environment Variables

- `NEXT_PUBLIC_PROD_URL`: The backend's Google Cloud URL, stored in Vercel settings to enable frontend-backend communication.

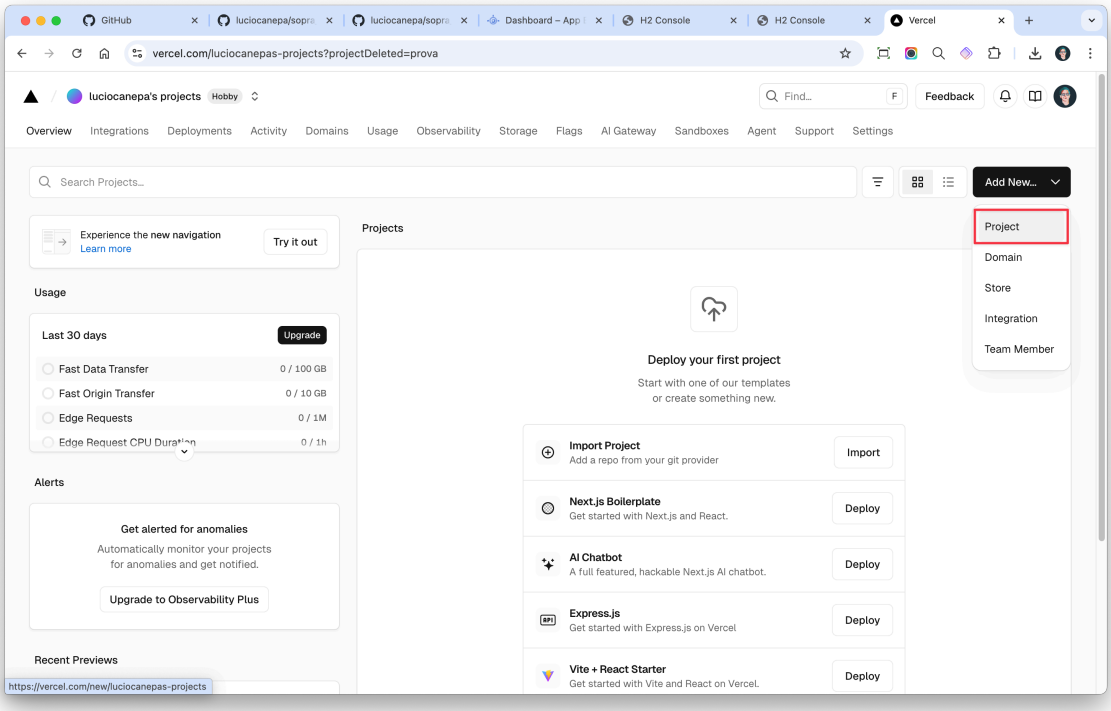
Set-up Vercel project and connect the client repository

Navigate to vercel.com

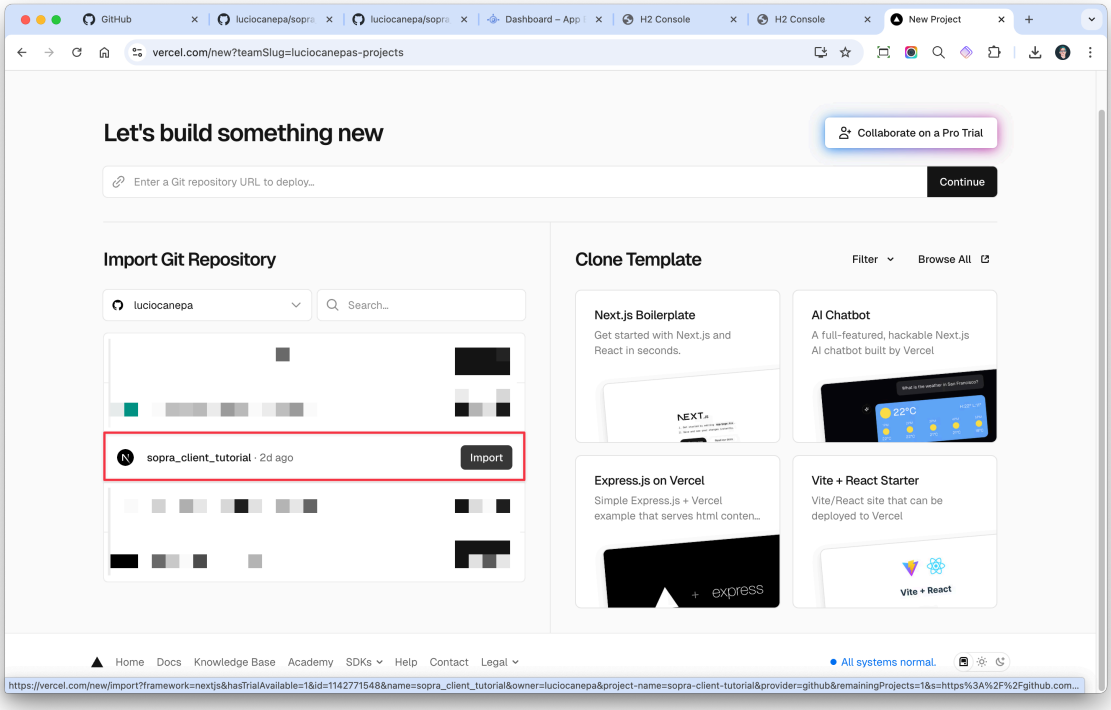
Create an account or login (best if with your Github account)



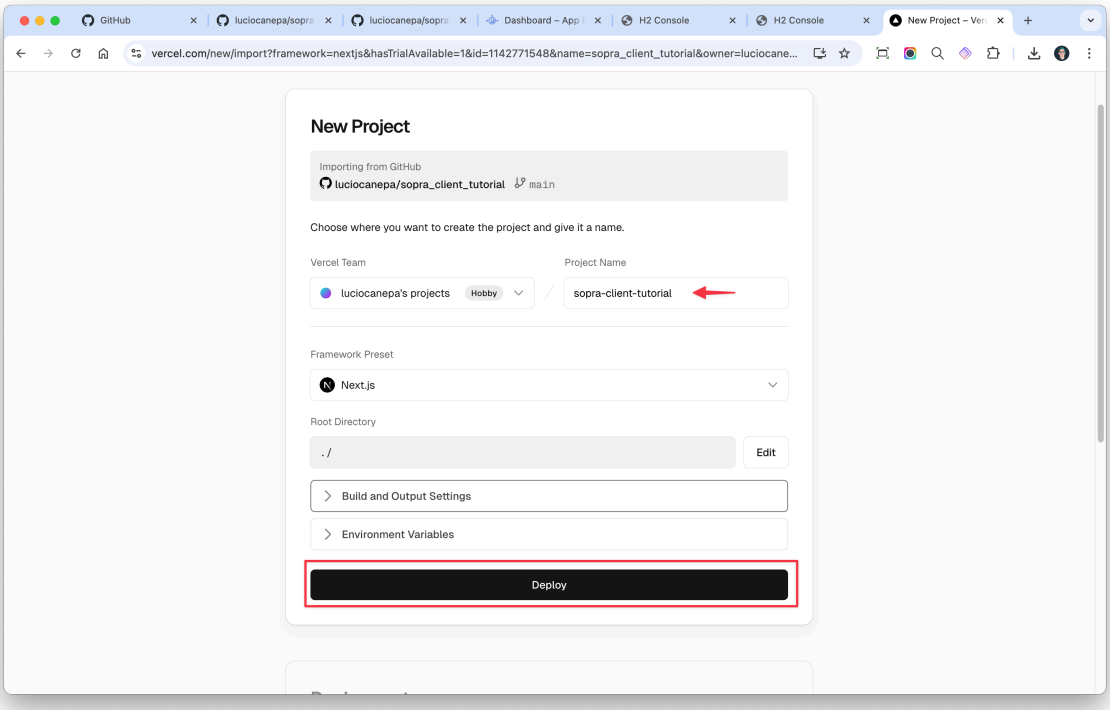
Create a new project



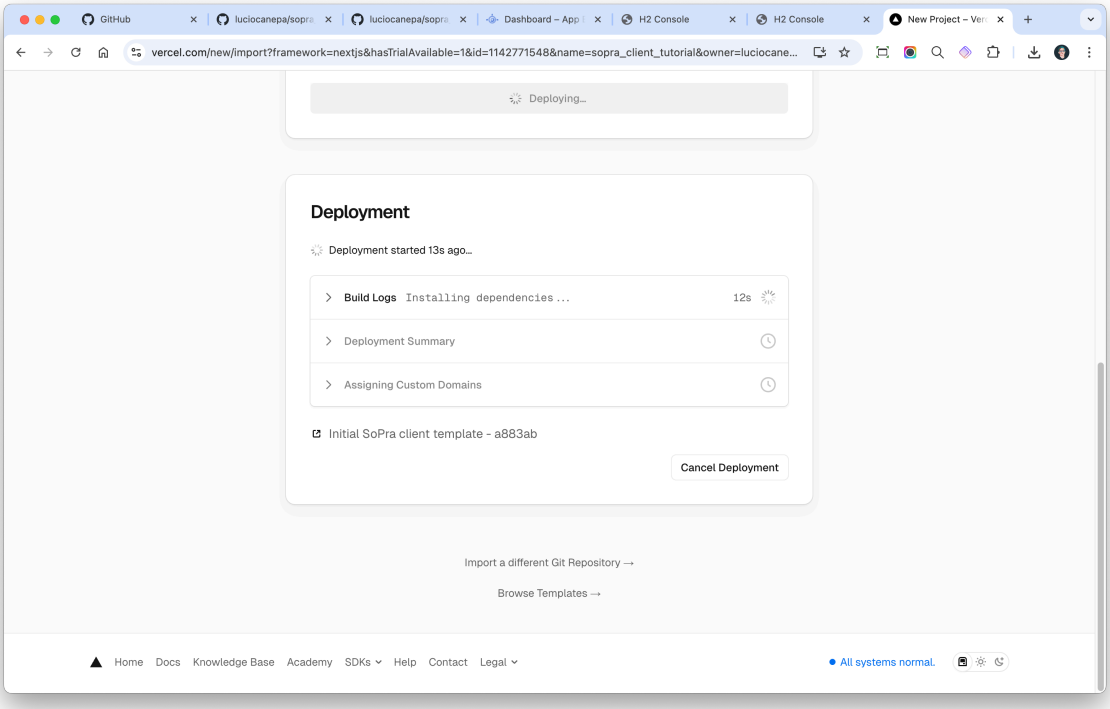
Connect the client repository (either you already connected your Github profile or you should do it now)



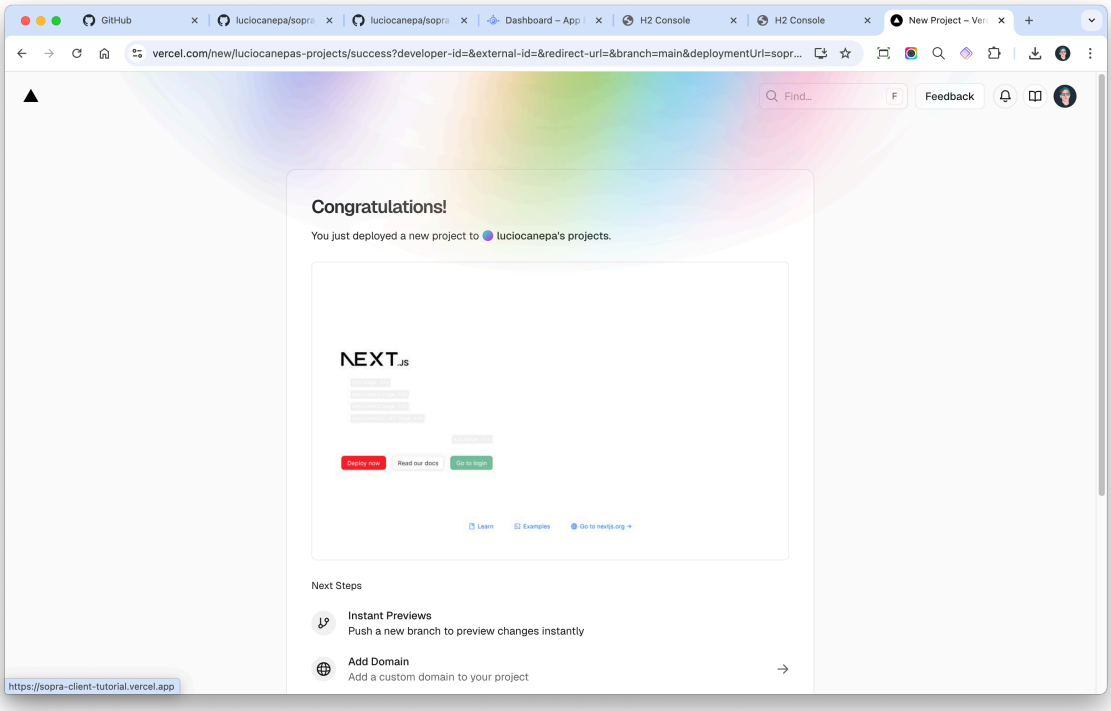
Name the project & click `Deploy`



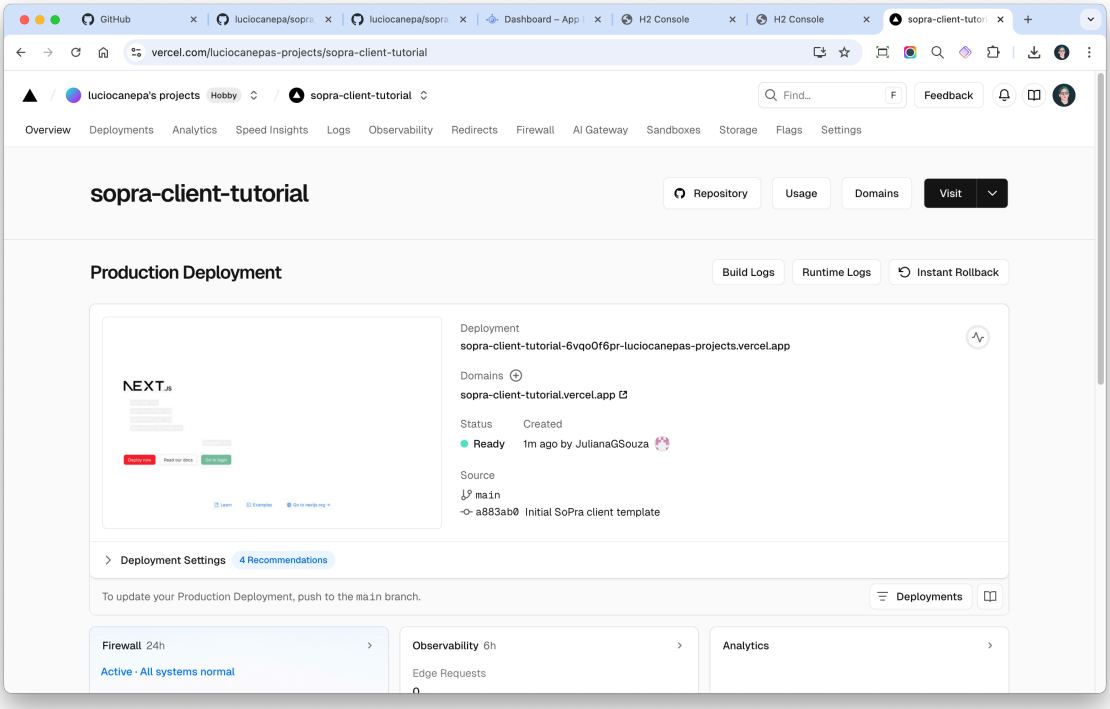
The deployment will start



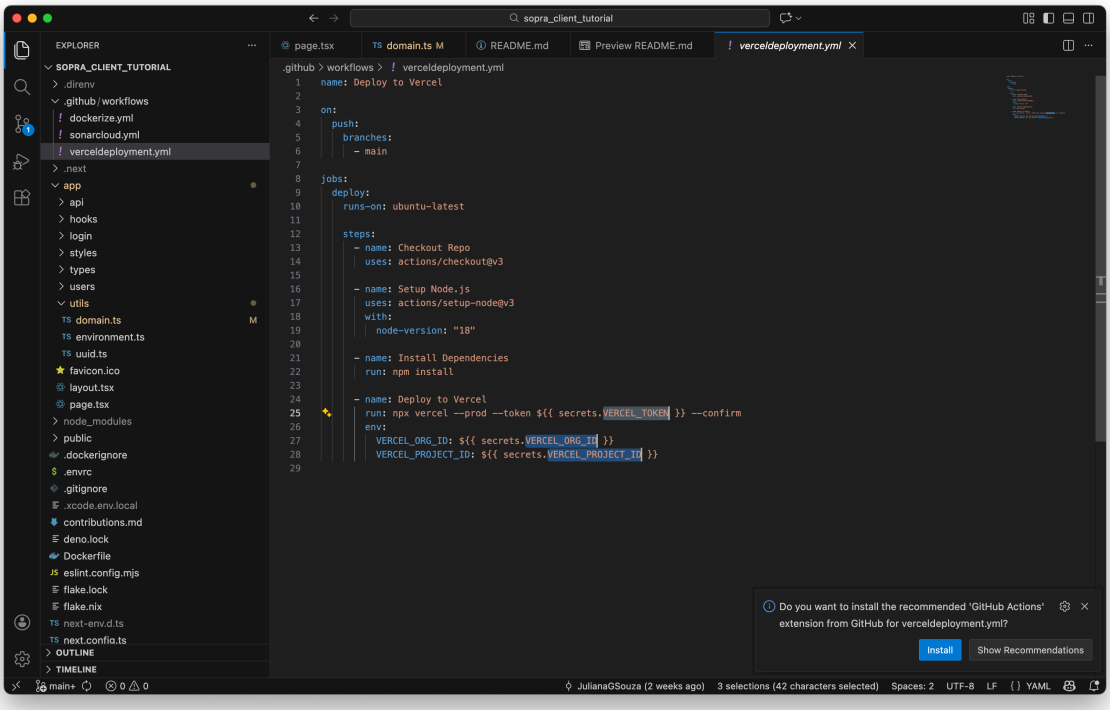
Confirmation: deployment was succesfull



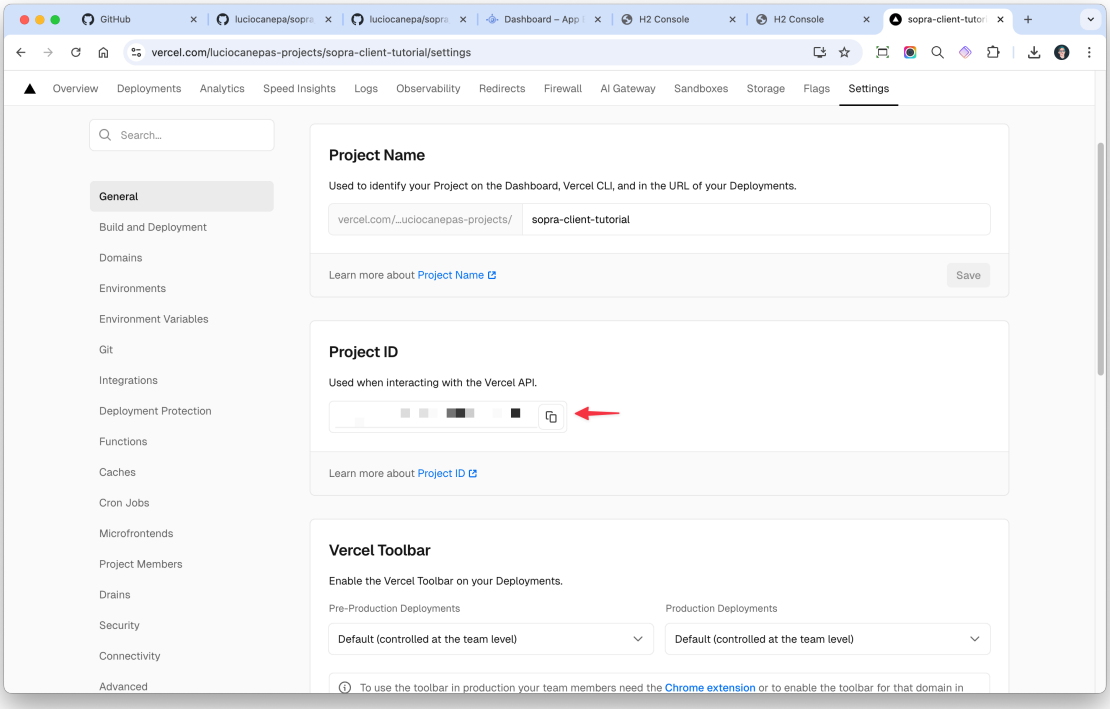
This is the overview page of your Vercel project



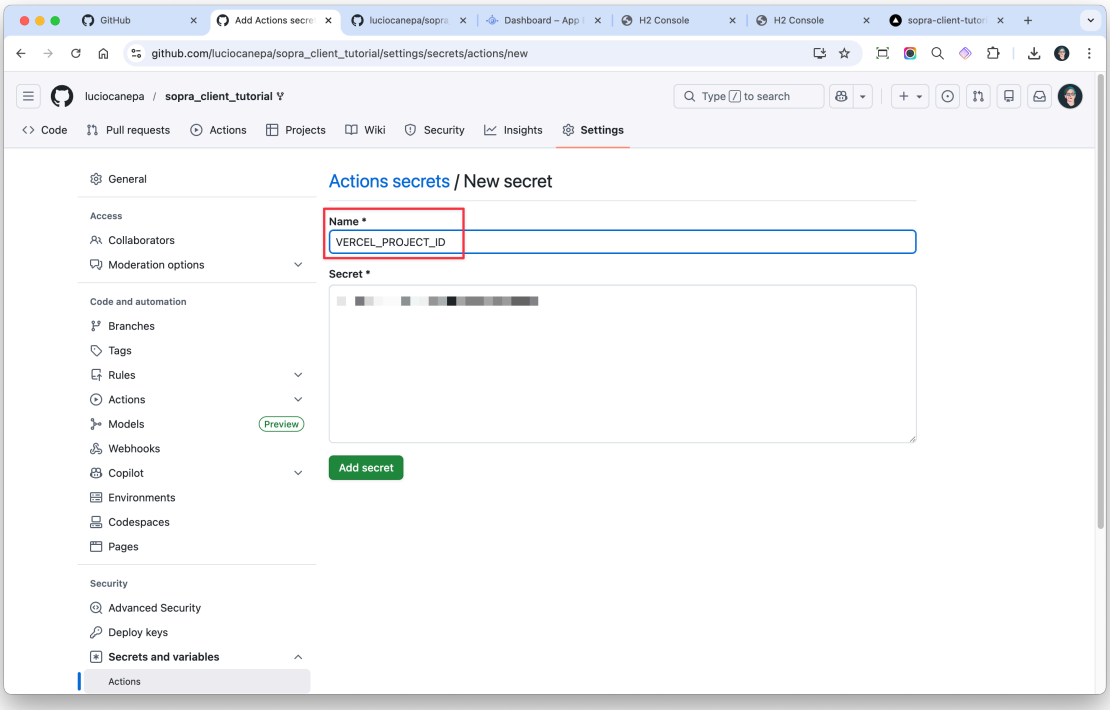
In the `.github/workflows/verceldeployment.yml` you can see that 3 secrets are needed: this should be added to the github client repository



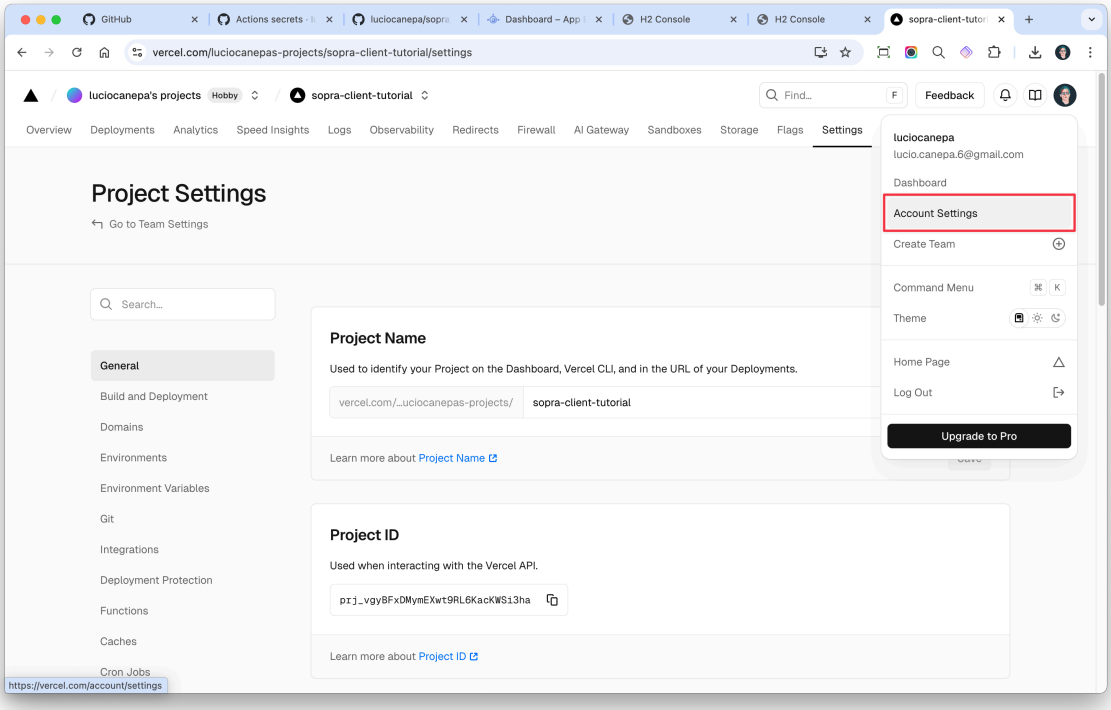
In Vercel under `Settings` > `General`, you will find your `Project ID` - copy it



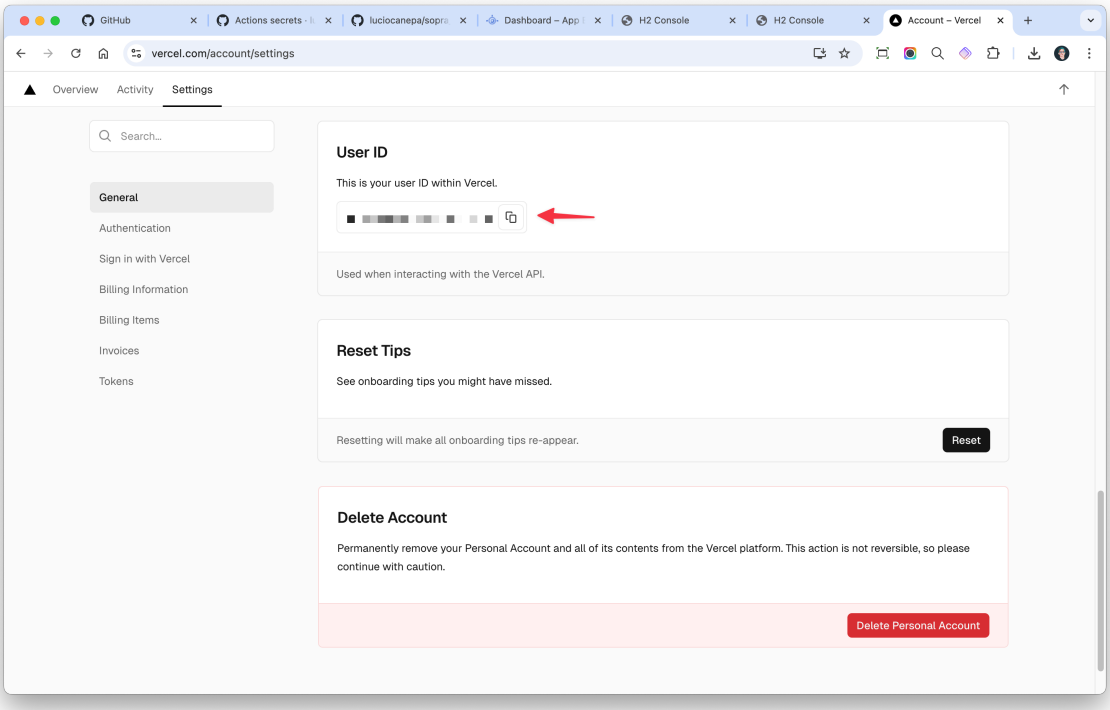
In your client repository, create an new repository secret matching precisely the name from your workflow file: `VERCEL_PROJECT_ID`



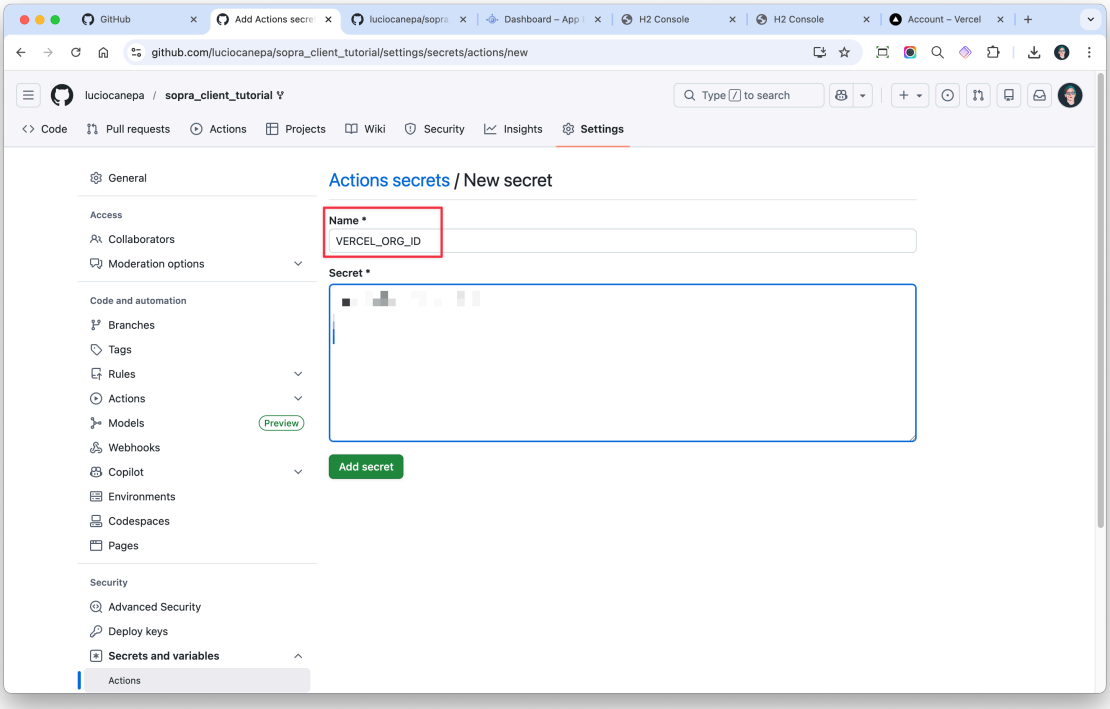
To find your `User ID` (= `Organization ID`), navigate to the `Account Settings`



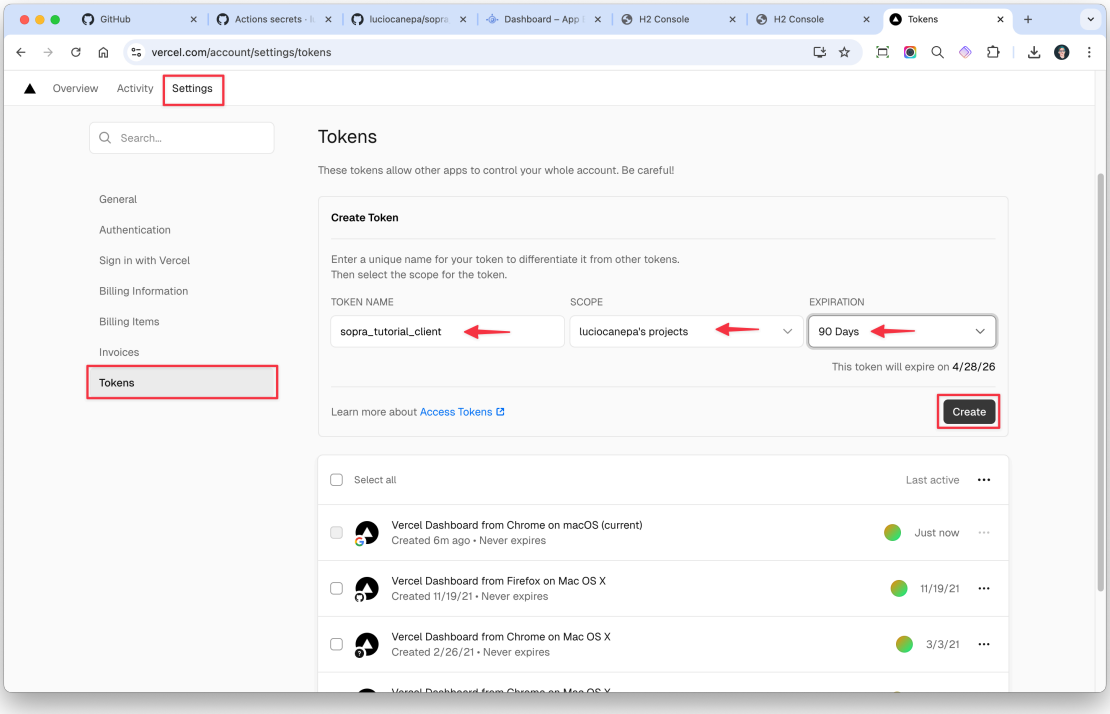
Under `General`, you find the `User ID` - copy it



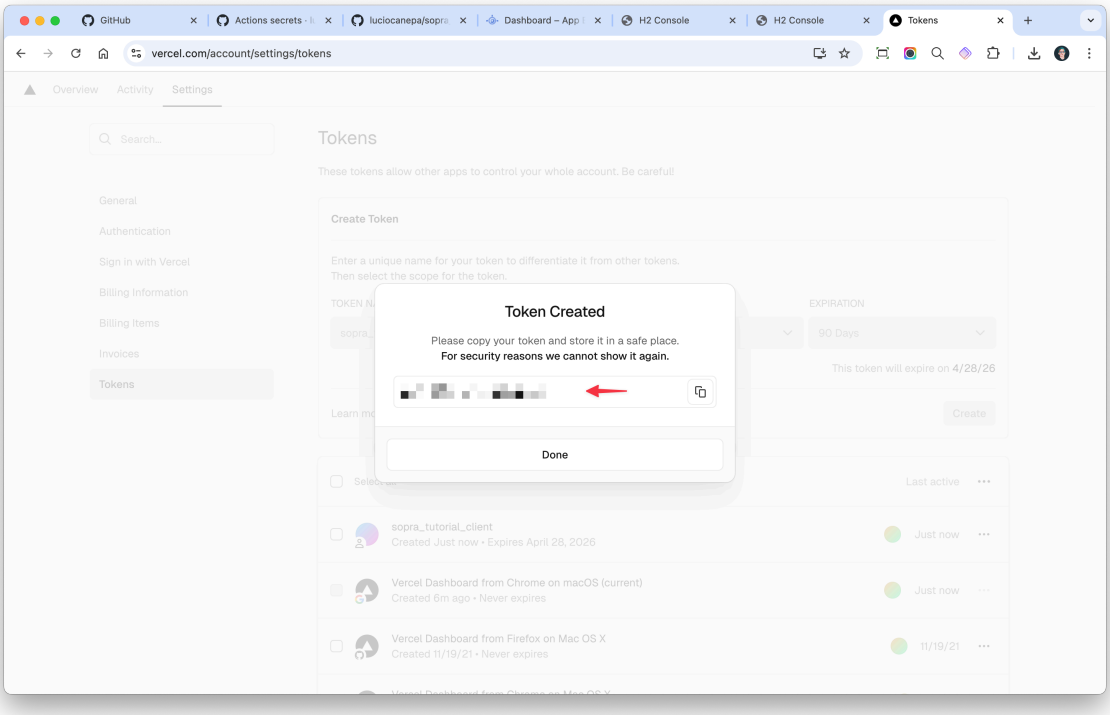
New repository secret `VERCEL_ORG_ID` - paste the copied `User ID` (= `Organization ID`)



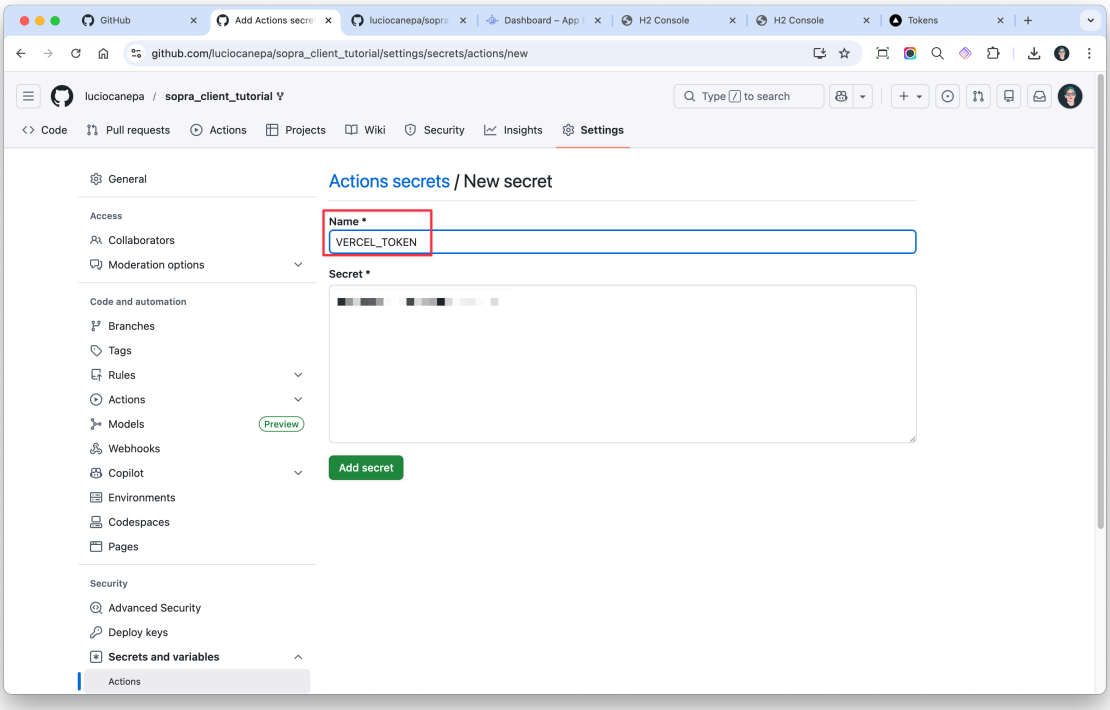
To generate a new `Token`, from the `Account Settings` > `Tokens` - give it a name (whatever you want), a scope (your organization) and an expiration date - click create



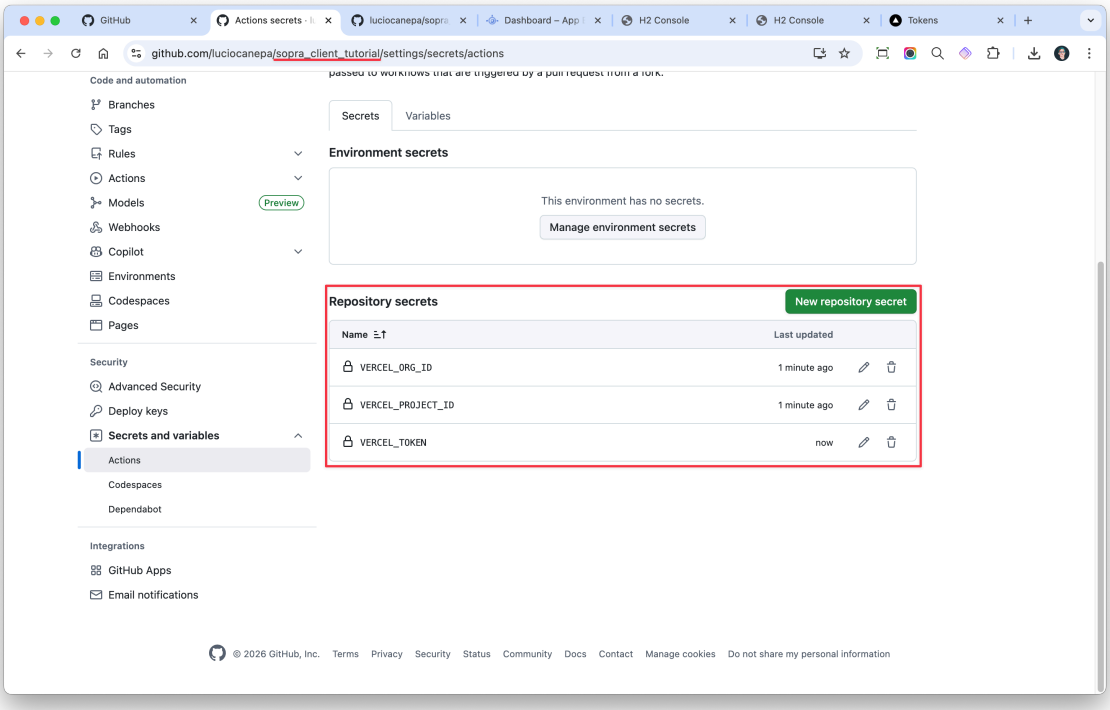
Confirmation: token is created - copy it



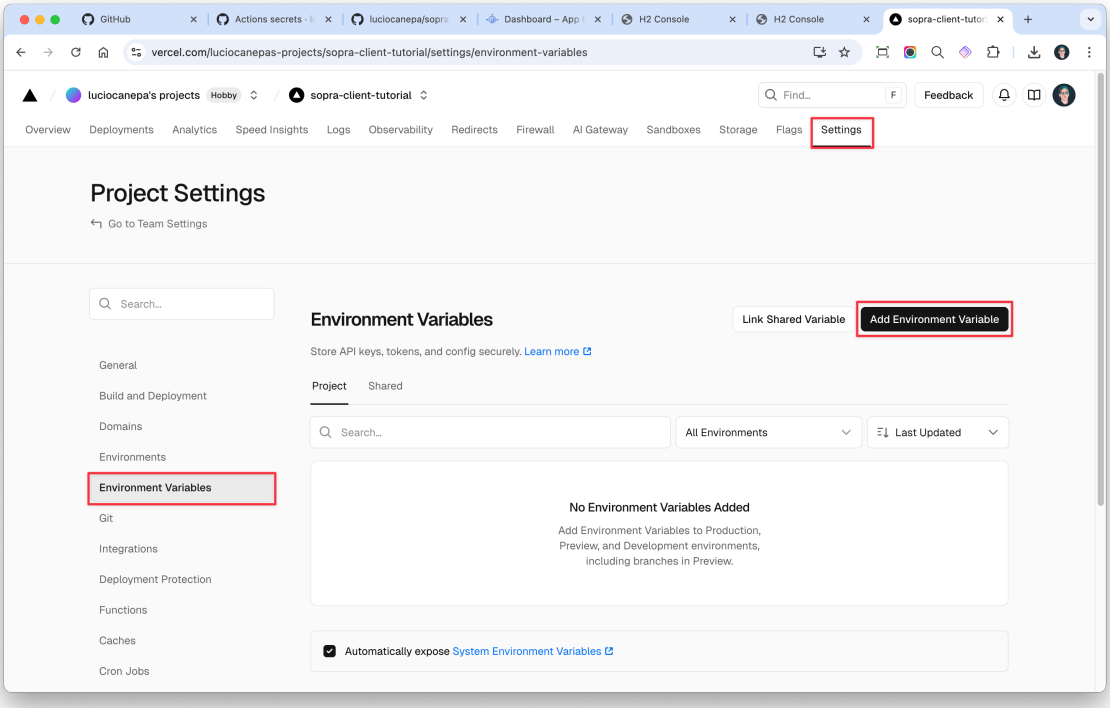
New repository secret: `VERCEL_TOKEN` - paste the copied token



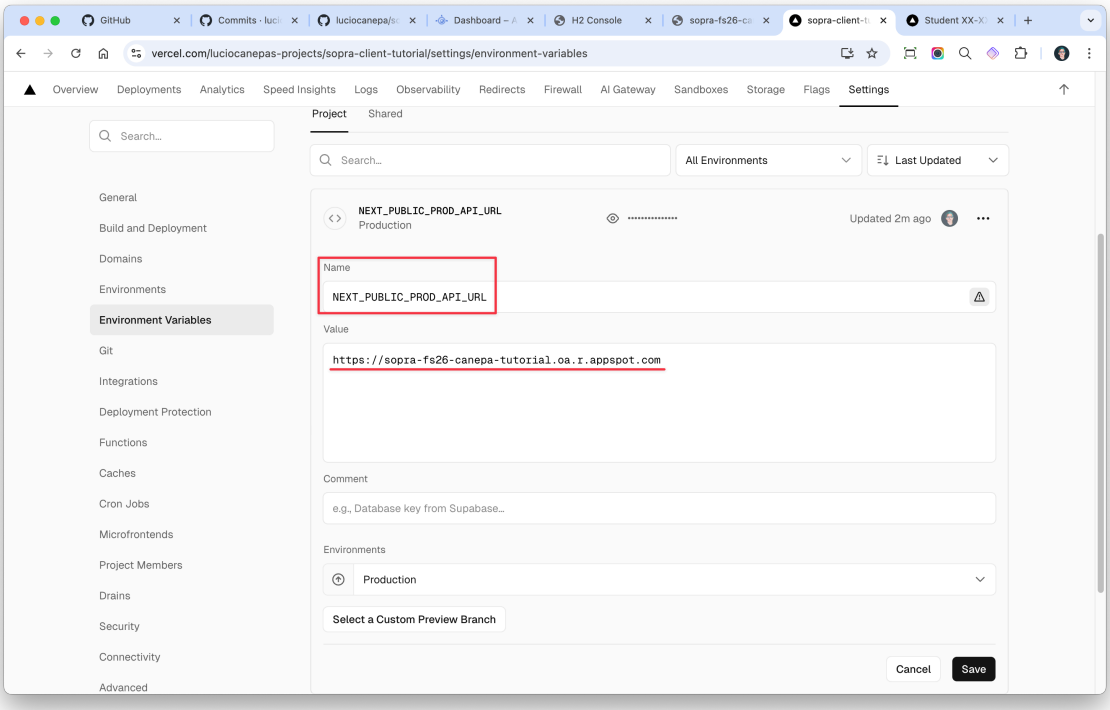
The 3 secrets should be visible under `Settings` > `Secrets and variables` in your **client repository**



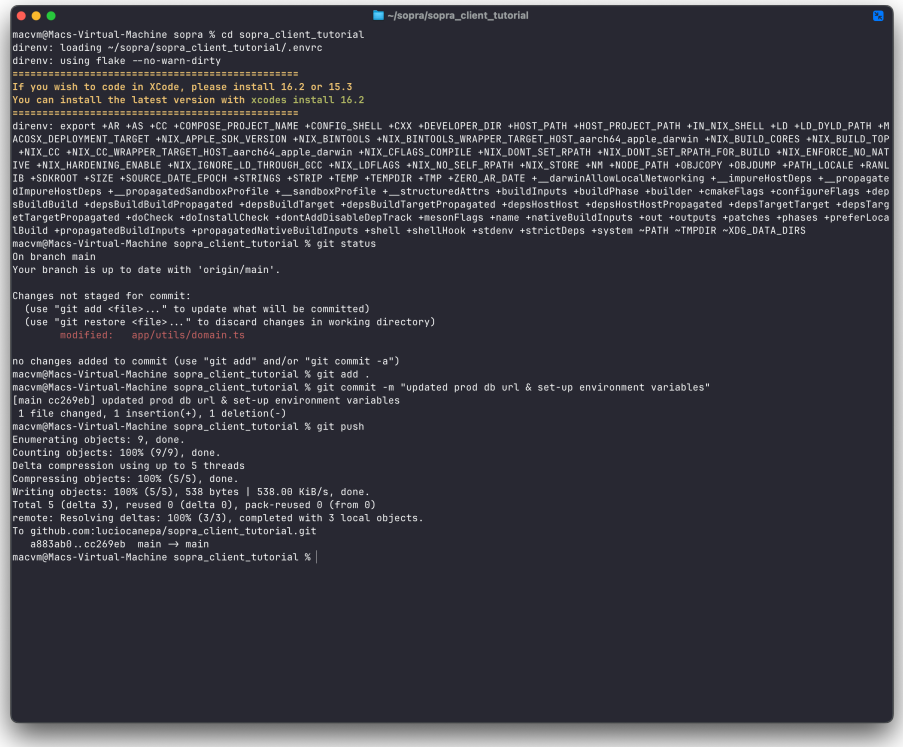
We need to add a last `Environment Variables` in Vercel > your client project > `Settings` > `Environment Variables` > `Add Environment Variable`



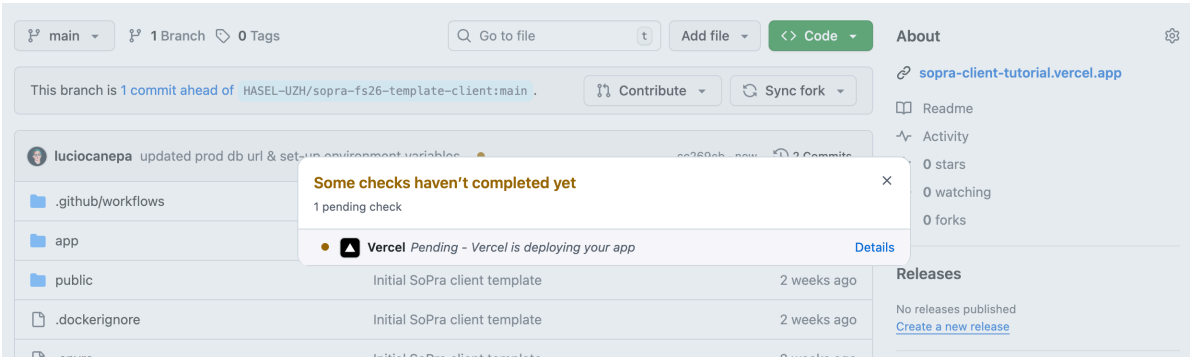
Name `NEXT_PUBLIC_PROD_URL` and paste the server production URL from Google Cloud console > `App Engine` (the same we pasted into the client repository in `domain.ts`)



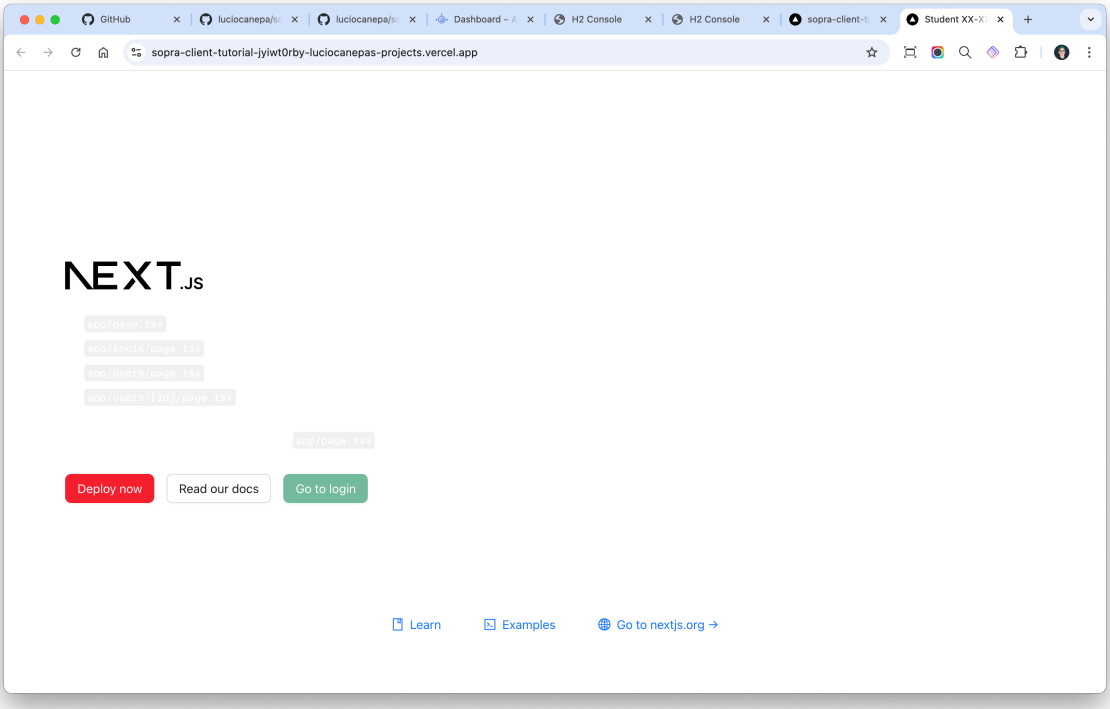
Now you can commit and push your changes from the client repository



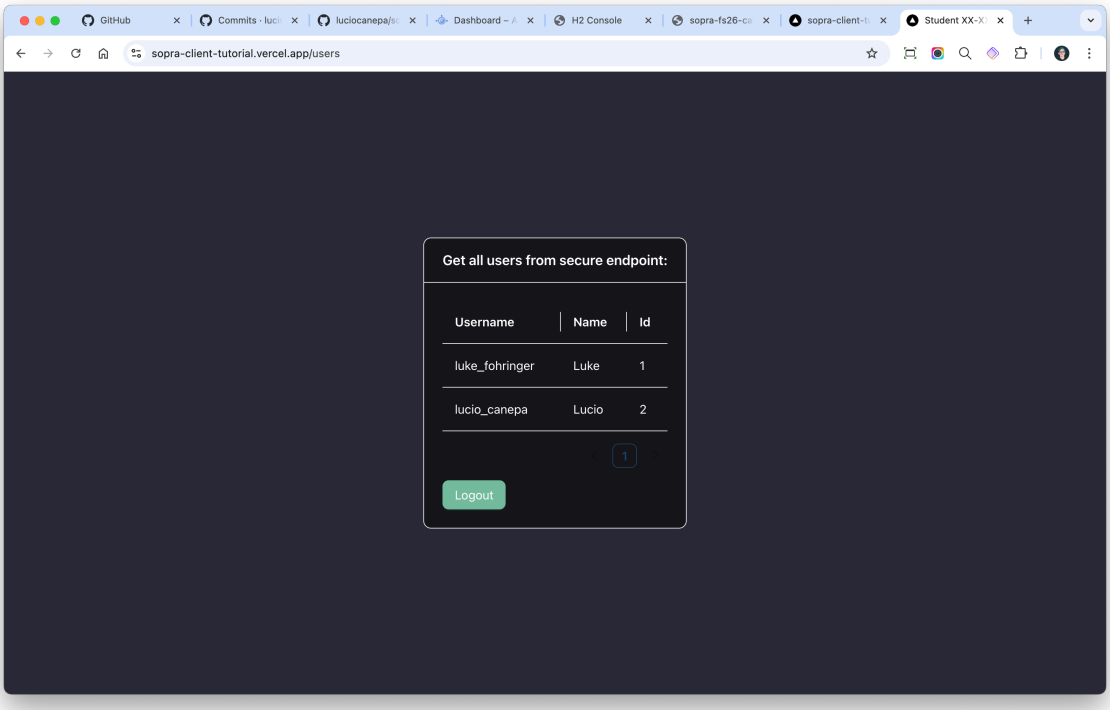
The Vercel deployment workflow will start



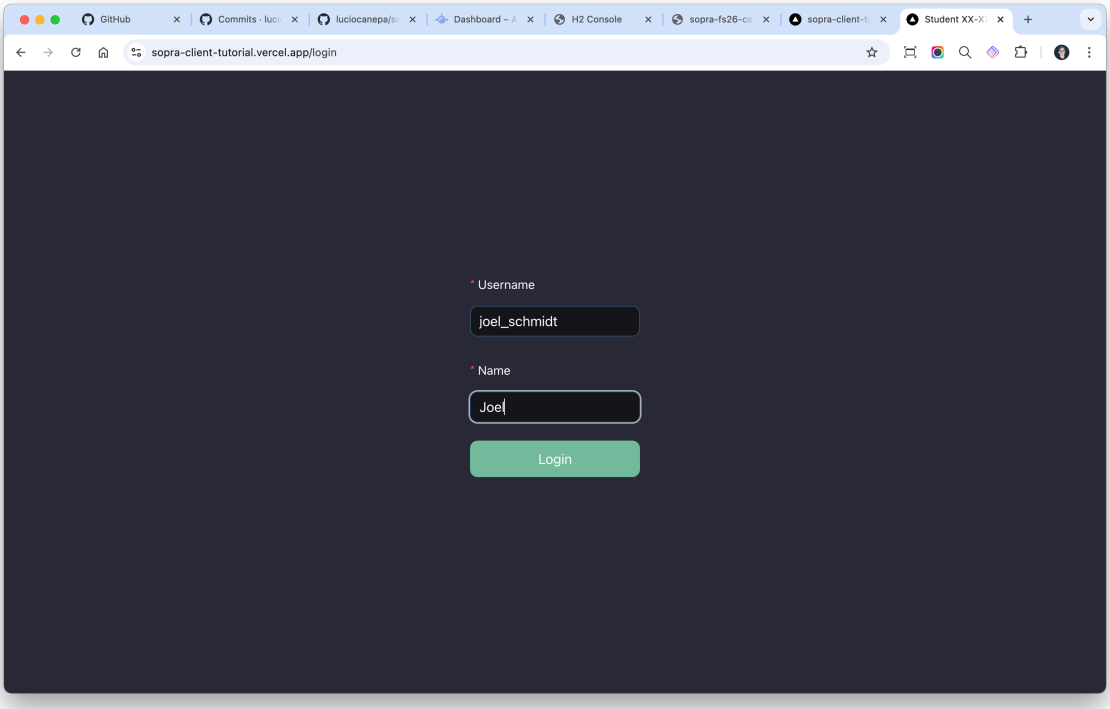
From the overview page of your project, you can `Visit` the deployed client side application



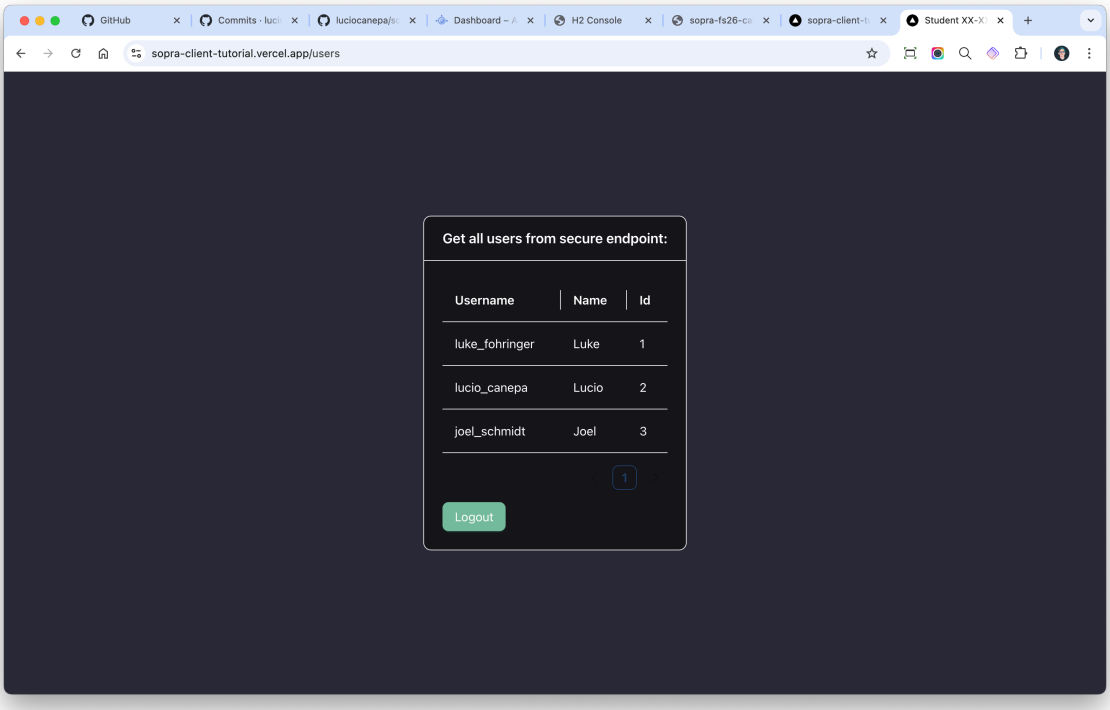
Verify it's connected with the backend (server repository deployed on Google Cloud) by navigating to `/users` - here we see the users we created before



under `/login` you can create a new user



and verify it has been created



5.3 Set-up Docker Hub and containerize your server application

Docker allows you to package your application into a *container* – a standardized unit that includes everything needed to run the software (code, runtime, libraries, dependencies). This makes it easy to run your application consistently on any machine.

Create Docker Account

1. Navigate to hub.docker.com and create an account
2. Incorporate your group number into the account name, for example sopra_group_12
3. After logging in, click Create repository
4. Name the repository *exactly the same* as your GitHub server repository name (e.g., sopra-fs26-template-server) – set visibility to Public & click Create

Adding Secrets

1. From Docker Hub, click on your profile picture > Account settings > Personal access tokens
2. Click Generate new token – give it a description (e.g., github-actions), select *Read & Write* access & click Generate
3. Copy the generated token immediately – you won't be able to see it again
4. Go to your *server repository* on GitHub > Settings > Secrets and variables > Actions

5. Click New repository secret and add the following 3 secrets:

Name	Value
dockerhub_username	Your Docker Hub username (e.g., sopra_group_12)
dockerhub_password	The PAT you generated (not your login password!)
dockerhub_repo_name	Your Docker Hub repository name

Caution

The secret names must match *exactly* what's in your workflow file, including lowercase letters and underscores.

Running Workflow

Push any change to your main branch to trigger the workflow:
bash git add . git commit -m "trigger docker build" git push origin main
Go to your GitHub repository > Actions tab – you should see the Dockerize workflow running.
Once successful (green checkmark), go to your Docker Hub repository – you should see your image with the latest tag.