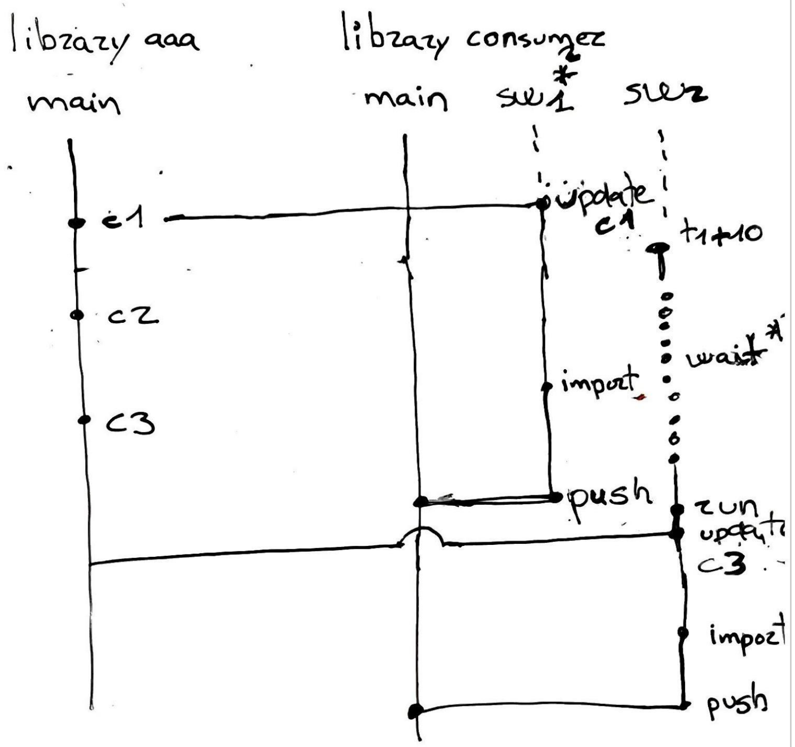
Update the library

From issue: https://github.com/Nautilus-Cyberneering/chinese-ideographs-website/issues/19

This document describes different problems and solutions to update a library (dvc git repo) in a consumer project (website).

Solution 1

Name: non-deterministic schedule workflow with concurrency option

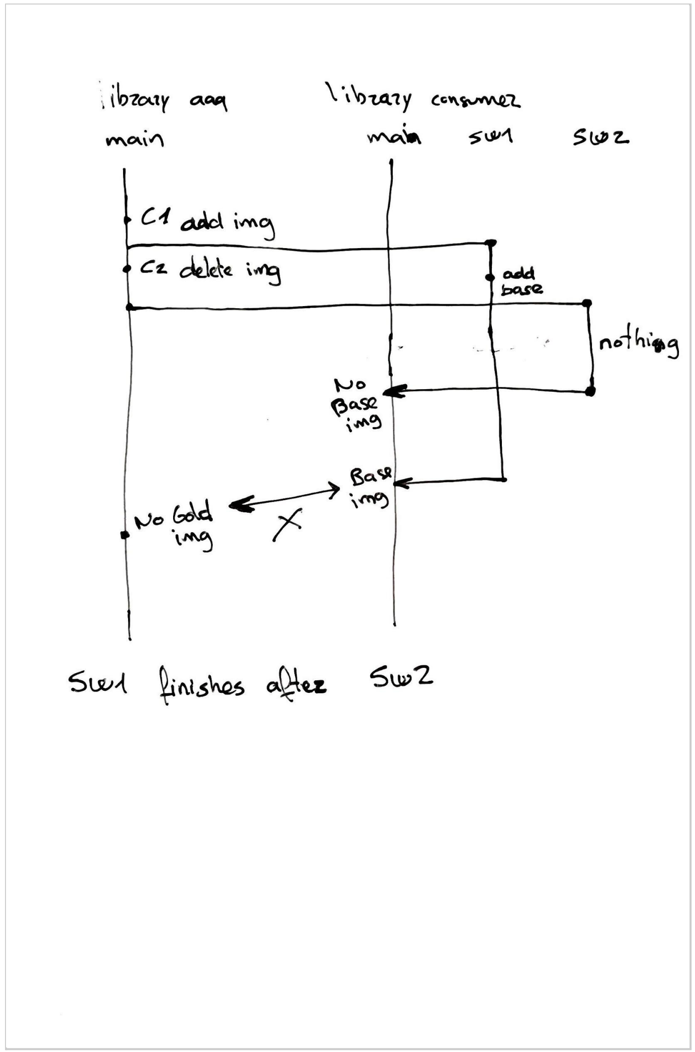


*1 Github "concurrency" attribute

*2 Scheduled workflow

Description:

- There is one scheduled workflow to update the library
- It's executed every 10 minutes
- It's similar to what you would do to update a software dependency manually. For example python package.
- It's called non-deterministic because the starting point it's not always the same. We are not processing a given commit. We get the latest library version when the workflow starts.
- When the sw1 (scheduled workflow 1) runs:
  - It updates the submodule from the current commit to the latest one.
  - It imports the images by creating new commits
  - It pushes the new commits into the main branch (all or nothing)
- If the ***sw1*** takes more than 10 minutes and the second workflow starts we can force it to wait with the GitHub workflow attribute "concurrency". That option prevents workflows or jobs to run at the same time if they have the same concurrency group. We are using the workflow name so we only run one "update library" workflow at the same time.
- Without the "concurrency" workflow attribute this solution would not be possible. We will get duplicate commits or even inconsistencies.


library aaq
main
library consumer
main
sw1
sw2
C1 add img
C2 delete img
add base
nothing
No Base img
Base img
No Gold img
Sw1 finishes after Sw2

Cases:

- ***Sw1*** succeeds:
    - ***Sw2*** will update the library from c2 to c3.
- ***Sw1*** fails:
    - ***Sw2*** will update the library from c1 to c3.

Pros:
- Easy solution for the problem we have right now.

Cons:
- We have to update the library and do all the secondary tasks (import Base images, transform them, …) in the same workflow. We have a workflow time limit. So this option is not valid if updating the library implies a lot of processing.
- We spend more time running workflows because if something fails we have to run the whole workflow again after fixing it.
- When a workflow fails work is accumulated for the next workflow. IF the library is updated very often and the workflows also fail often the workflow work becomes bigger and bigger, because the same workflow has to process all the pending changes.
- We force some merges to wait until other errors are fixed. For example, if we add a Gold image in the library and that causes the workflow to fail, but then there are 3 more commits with more Gold images, the newer ones are not going to be merged until we fix the problem with the first one.

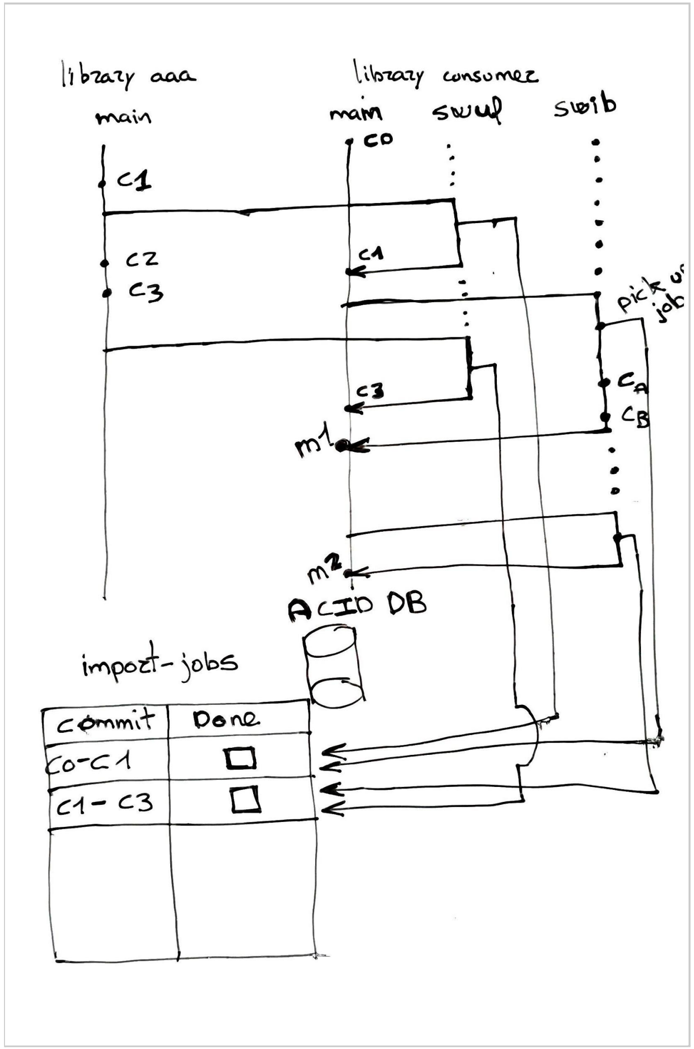
Sample workflow: https://github.com/josecelano/library-consumer/blob/main/.github/workflows/update-library-aaa-without-pr.yml

Solution 2

Name: deterministic schedule workflow to process jobs

Instead of processing all the pending changes since the last library update, we can try to schedule a workflow only for a library increment.

When we detect a change in the library (polling) we create a job that has to be done by a worker (another workflow). Job queue pattern.



commit	Done
c0-c1	☐
c1-c3	☐

Description:

- We need a first scheduled workflow (***swul*** -scheduled workflow update library) that checks every 10 minutes if there is a new library commit in the library. If there is a new commit it will:
    - Create an auto-commit to update the library submodule
    - Queue a new job to import the Base images
    - Both things have to be done atomically.
- We need a second scheduled workflow (***swib*** - scheduled workflow to import base images) that processes the jobs in the queue. If the are pending jobs it will:
    - Pick up a pending job.
    - Import the base images and create the commits.
    - The merge (push from runner main branch to origin) and marking the job as done has to be an atomic operation.
- We do not use pull requests, we work on the main branch in the runner and we directly push into the origin.

Pending:

This is an incomplete solution. There are some things I do not know how to solve yet:

- How we are going to implement the ACID DB.
    - If you use an external service then we have to ensure that the transaction in the DB and the merge are atomic.
    - We can try to create the DB only with git (using merge conflicts to implement transactions). This is the solution we discussed.

Json DB:

If we want to use only git the solution could be:

- A directory "import_jobs" containing one file per job (a record). It could be also one big json file with all records but that can lead to performance problems sooner than the other option.
- We run only one ***swul*** *workflow at a time.*
- We run only one ***swib*** *workflow at a time.*
- The ***swib*** workflow can only process the jobs in the order they were created

The simplest solution is both workflows running only one thread. If we want to run workflows in parallel we have to find a way to create a merge conflict when a second workflow is processing the same job (optimistic locking). One option could be to create an extra attribute in the json job with a version/token/checksum (encoded in base64 or hash) that depends on the workflow run id, for example. In order to produce a merge conflict even if the final job record has the same content (done=true)

Problem pending to solve:

If we run a finished workflow again (for example manually), the workflow is going to create duplicate commits because it's going to checkout the code in a commit where the job was not finished. The DB is in the code so the state changes depending on the commit. Maybe we can avoid this problem using the solution I proposed above (provoke a merge conflict with an optimistic looking field). The problem is even if the workflow is not executed in parallel you can execute the same workflow twice

Pros:

- We can decouple the library update from the tasks we have to do. If we need to do one more thing (not only importing Base images), we can easily extend the design.
- We do not make tasks bigger. The next task is not processed until we finish the first one, so workflows do not become bigger and bigger.

Cons:

- We still have a bottleneck. If there is a problem in a previous job we can not execute the next one. But that's an intrinsic problem we cannot solve. We have to process the library commits in the order they are created.

IMPORTANT:

- This solution does not use pull requests
- A more general or complex solution can be implemented if we want general jobs not only for this case.

Alternatives

- Try to implement a message queue or ACID database with git content or metadata:
    - Example https://github.com/emad-elsaid/gitmq
    - I tried to implement here an event store but you end up implementing something complex that has already been solved and it's very tricky to implement.
        - I could store the events in a folder
        - But I need a workflow to create the job and then another workflow to process the job.
- Using pull requests:
    - We can create new PR easily but there are 2 problems:
        - PR generated automatically do not execute workflows
        - There is no easy way to auto-merge the PRs. We need another workflow to heck when a PR is ready to merge.
- Using an external message queue provider:
    - Workflows can send and consume events from that service.
    - It requires setting up one more external service. We already need to set up the DVC storage.
    - It can make testing harder.
- Create a GitHub App
    - The hosted app could use jobs/queues or whatever it needs and create the PR and merge them.
    - This option seems to be too complex.