deploy2zenodo

Daniel Mohr

2024-10-01

Abstract

deploy2zenodo is a script to deploy your data to zenodo providing automatic workflow via ci pipeline

Contents

deploy2zenodo **1**
- preamble . . . 1
- intention . . . 2
- how-to . . . 2
  - simple workflow . . . 2
  - very simple workflow . . . 4
  - triggered workflow . . . 5
  - complex workflow . . . 6
  - very complex workflow . . . 7
- script parameter . . . 7
  - DEPLOY2ZENODO_API_URL . . . 8
  - DEPLOY2ZENODO_ACCESS_TOKEN . . . 8
  - DEPLOY2ZENODO_DEPOSITION_ID . . . 8
  - DEPLOY2ZENODO_JSON . . . 8
  - DEPLOY2ZENODO_UPLOAD . . . 9
  - DEPLOY2ZENODO_SKIP_PUBLISH . . . 10
  - DEPLOY2ZENODO_DRYRUN . . . 10
  - DEPLOY2ZENODO_SKIPRUN . . . 10
  - DEPLOY2ZENODO_SKIP_NEW_VERSION . . . 10
  - DEPLOY2ZENODO_GET_METADATA . . . 10
  - DEPLOY2ZENODO_SKIP_UPLOAD . . . 11
  - DEPLOY2ZENODO_CURL_MAX_TIME . . . 11
  - DEPLOY2ZENODO_CURL_MAX_TIME_PUBLISH . . . 11
  - DEPLOY2ZENODO_ADD_IsCompiledBy_DEPLOY2ZENODO . . . 11
  - DEPLOY2ZENODO_ADD_IsNewVersionOf . . . 11
  - DEPLOY2ZENODO_ADD_IsPartOf . . . 12
- CI pipeline . . . 13
- script . . . 13
- harvesting . . . 14
- curating . . . 14
- license: Apache-2.0 . . . 14

deploy2zenodo

preamble

deploy2zenodo is a shell script to deploy your data to zenodo. You can use it in a CI pipeline as

an automatic workflow.

Environmental variables allow very flexible use. Depending on the selected flags, the data can be curated before deployment in a merge request, in the zenodo web interface or not curated at all.

intention

To satisfy the FAIR[1] principles[2], publications should be deployed to an open repository. In this way the publication gets a PID (persistent identifier) and at least the metadata is publicly accessible, findable and citable. Furthermore, current discussions about KPIs (key performance indicator) for software and data publications also lead to the need to generate PIDs for software and data.

Especially software usually is not citable by a PID. To overcome this and make software academically significant we provide here a tool for automatic publication to the open repository zenodo.

In principal the same is true for all kind of scientific data (e. g. measurements, software and results such as papers). For every data managed in a version control system an automatic publication to an open repository is useful[3].

Software in particular is subject to frequent changes, resulting in many versions. This leads to the urge to automate the publishing process. This is not only about making the software usable through software repositories, but also about the citability of individual versions.

how-to

There are many possibilities to use `deploy2zenodo` but in this how-to section we will focus on a few typically use cases.

simple workflow

This workflow reflects the primary focus of `deploy2zenodo`.

Go to your zenodo account and create an access token.

Store it in a GitLab CI/CD variable as `DEPLOY2ZENODO_ACCESS_TOKEN`. Use the flags Mask variable and Protect variable. Keep in mind the token is sensitive and private information. Therefore you should not share it or make it public available.

Then the GitLab CI/CD pipeline could look like (we use here sandbox.zenodo.org instead of zenodo.org for testing purpose):

```yaml
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/

prepare_release_and_deploy2zenodo:
  stage: build
  image:
    name: alpine:latest
  variables:
    DEPLOY2ZENODO_JSON: "mymetadata.json"
  script:
    # prepare
    - TAG=$(grep version library.properties | cut -d "=" -f 2)
    - |
      echo '{"metadata":{"creators":[{"name":"family, given"}],\
        "license":{"id":"GPL-3.0-or-later"},"title":"test script alpine",\
```

[1] FAIR Principles
[2] An interpretation of the FAIR principles to guide implementations in the HMC digital ecosystem.
[3] Guidance on Versioning of Digital Assets.

```
        "version":"***","upload_type":"software"}}' | \
        jq ".metadata.version = \"$TAG\"" | tee "$DEPLOY2ZENODO_JSON"
    # prepare release
    - echo "DESCRIPTION=README.md" > variables.env
    - echo "TAG=$TAG" >> variables.env
    # prepare deploy2zenodo
    - echo "DEPLOY2ZENODO_JSON=$DEPLOY2ZENODO_JSON" >> variables.env
    - DEPLOY2ZENODO_UPLOAD="v$TAG.zip"
    - git archive --format zip --output "$DEPLOY2ZENODO_UPLOAD" "$TAG"
    - echo "DEPLOY2ZENODO_UPLOAD=$DEPLOY2ZENODO_UPLOAD" >> variables.env
  artifacts:
    reports:
      dotenv: variables.env
    paths:
      - $DEPLOY2ZENODO_JSON

release_job:
  stage: deploy
  rules:
    - if: $CI_COMMIT_TAG
      when: never
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  image:
    registry.gitlab.com/gitlab-org/release-cli:latest
  script:
    - cat /etc/os-release
  release:
    name: 'v$TAG'
    description: '$DESCRIPTION'
    tag_name: '$TAG'
    ref: '$CI_COMMIT_SHA'

deploy2zenodo:
  rules:
    - if: $CI_COMMIT_TAG
      when: never
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  variables:
    DEPLOY2ZENODO_API_URL: "https://sandbox.zenodo.org/api"
    DEPLOY2ZENODO_DEPOSITION_ID: "create NEW record"
```

We use here 3 jobs:

- The job `prepare_release_and_deploy2zenodo` prepares the variables and data for the following jobs. You can choose how to get the variables and data from your project/repository. (see hints in DEPLOY2ZENODO_JSON and DEPLOY2ZENODO_UPLOAD)
- The job `release_job` uses the workflow Create release metadata in a custom script.
- The job `deploy2zenodo` publishes the data to zenodo.

The variables are passed between the jobs using dotenv variables. And the data are passed using job artifacts.

After the first run of the above pipeline (job `deploy2zenodo`) adapt `DEPLOY2ZENODO_DEPOSITION_ID` to store the record id. Only then you are able to release new versions to zenodo.

In this example, `prepare_release_and_deploy2zenodo` always runs while the other jobs only run when the default branch is changed. This makes it possible to check the artifacts during a merge

request.

The used environment variables (see script parameter) can be provided in many different ways as a GitLab CI/CD variable, e. g.:

- CI/CD variable in the UI
  - not stored in the repository
  - possible to Mask variable
  - possible to Protect variable
  - used for private data (e. g. access token)
- CI/CD variable in the .gitlab-ci.yml
  - stored in the repository
  - in public projects also publicly accessable

You should think about which information to store at which place. Here a few simple considerations:

variable	private data	note
DEPLOY2ZENODO_API_URL	no	Should a user find your publication?
DEPLOY2ZENODO_ACCESS_TOKEN	YES	Should not be shared with anyone!
DEPLOY2ZENODO_DEPOSITION_ID	no	Should a user find your publication?
DEPLOY2ZENODO_JSON	?	Is the publication public?
DEPLOY2ZENODO_UPLOAD	?	Is the publication public?

Sometimes it is easier to change the variable in the UI. For example in your first step you should set `DEPLOY2ZENODO_API_URL="https://sandbox.zenodo.org/api"` and `DEPLOY2ZENODO_DEPOSITION_ID="create NEW record"` to initiate and test your pipeline. After success you should change to `DEPLOY2ZENODO_API_URL="https://zenodo.org/api"`. And after you have created your first record, also change `DEPLOY2ZENODO_DEPOSITION_ID` to the returned value to update your dataset next time (and not create a new one). If you store these variables in the user interface, you can change them without touching your repository. On the other hand, the metadata provided via `DEPLOY2ZENODO_JSON` and the data provided via `DEPLOY2ZENODO_UPLOAD` may be created dynamically and it could therefore make sense to create these variables dynamically as well.

There are also optional variables that can help to adapt the workflow to the the individual use case. For example, DEPLOY2ZENODO_SKIP_PUBLISH allows you to curate the upload to zenodo in the zenodo web interface before publishing. This is especially useful if you are setting up the workflow for the first time in your own project – but can also be used at any time.

An example test project is deploy2zenodo_test_simple_workflow_update.

very simple workflow

It is not necessary to create a release for publication. But we think this is the typically use case for software publication.

For a very simple workflow running when creating a tag, you could use something like:

```yaml
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/

deploy2zenodo:
  stage: deploy
  rules:
```

```
    - if: $CI_COMMIT_TAG
  variables:
    DEPLOY2ZENODO_API_URL: "https://sandbox.zenodo.org/api"
    DEPLOY2ZENODO_JSON: "CITATION.json"
    DEPLOY2ZENODO_DEPOSITION_ID: "create NEW record"
    DEPLOY2ZENODO_UPLOAD: "$CI_PROJECT_NAME-$CI_COMMIT_TAG.zip"
    DEPLOY2ZENODO_ADD_IsCompiledBy_DEPLOY2ZENODO: "yes"
    DEPLOY2ZENODO_ADD_IsNewVersionOf: "yes"
    DEPLOY2ZENODO_ADD_IsPartOf: "yes"
    DEPLOY2ZENODO_GET_METADATA: "result.json"
  before_script:
    - env
    - echo https://dl-cdn.alpinelinux.org/alpine/edge/community >> /etc/apk/repositories
    - apk add --no-cache cffconvert curl git jq
    - publication_date=$(echo "$CI_COMMIT_TIMESTAMP" | grep -Eo "^[0-9]{4}-[0-9]{2}-[0-9]{2}")
    - |
      cffconvert -i CITATION.cff -f zenodo | \
        jq -c '{"metadata": .} | .metadata += {"upload_type": "software"}' | \
        jq -c ".metadata.related_identifiers += [
          {\"relation\": \"isDerivedFrom\",
          \"identifier\": \"$CI_SERVER_URL/projects/$CI_PROJECT_ID\"}] |
          .metadata.version = \"$CI_COMMIT_TAG\" |
          .metadata.publication_date = \"$publication_date\"" | \
        tee "$DEPLOY2ZENODO_JSON" | jq -C .
    - git archive --format zip --output "$DEPLOY2ZENODO_UPLOAD" "$CI_COMMIT_TAG"
  artifacts:
    paths:
      - $DEPLOY2ZENODO_JSON
      - $DEPLOY2ZENODO_GET_METADATA
```

Such a simple workflow uses deploy_deploy2zenodo_to_zenodo in the job `deploy2zenodo` to publish itself.

triggered workflow

In many projects there is more than one maintainer. Therefore it is not possible to store the user token for zenodo as CI variable in the project. Otherwise, the user token would be shared with the other maintainers.

Using this triggered workflow allows to restrict the use of the user token to a specific zenodo record for other maintainers.

But the project `A` with more than one maintainer can trigger a pipeline in another (private) project `B` with only one maintainer, e. g.:

```
trigger:
  stage: .post
  rules:
    - if: $CI_COMMIT_TAG
      when: never
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  image:
    name: alpine:latest
  script:
    - apk add --no-cache curl
    - curl -X POST --fail -F token="$TRIGGER_TOKEN" -F ref=main "$TRIGGER_URL"
```

Storing the `TRIGGER_TOKEN` as protected and masked CI variable in project `A` allows any maintainer to use it and trigger the pipeline.

In the project `B` you can use deploy2zenodo as normal, e. g.:

```yaml
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/

prepare_deploy2zenodo:
  image:
    name: alpine:latest
  script:
    - PROJECT_A_REPO=$(mktemp -d)
    - git clone --branch main --depth 1 "$PROJECT_A_URL"
    - |
      (cd "$PROJECT_A_REPO" && \
       git archive --format zip -o "$DEPLOY2ZENODO_UPLOAD" \
       "$(git tag | sort -t "." -n -k 3 | tail -n 1)")
  artifacts:
    expire_in: 1 hrs
    paths:
      - $DEPLOY2ZENODO_UPLOAD

deploy2zenodo:
  variables:
    DEPLOY2ZENODO_DEPOSITION_ID: "create NEW record"
    DEPLOY2ZENODO_API_URL: "https://sandbox.zenodo.org/api"
```

There are various ways to trigger a pipeline, e. g:

- trigger a pipeline by trigger token
- trigger using Multi-project pipelines

In the CI pipeline above the token method is used. In the CI pipeline of deploy2zenodo the multi-project pipeline is used.

Be careful: The trigger job from project `A` may overwrite variables in the triggered job from project `B`. This could lead to security concerns. Maybe Restrict who can override variables could help to overcome this.

More details: In project `A` something exists that should be published on zenodo. In project `B` the content of project `A` is published on zenodo. The pipeline in project `B` can be triggered so that this happens automatically when corresponding changes are made in project `A` (e. g. merge to default branch). Project `B` should rely as little as possible on project `A`. Unfortunately, variables can be transferred when triggering (from project `A`) and these are not trustworthy. For example, a maintainer from project `A` could pass `DEPLOY2ZENODO_API_URL` in this way and thus force communication to another server. This could cause the user token to be leaked. However, it is no problem to save the user token in project `B` as CI variable `DEPLOY2ZENODO_ACCESS_TOKEN`. This variable could then be overwritten from project `A`, but not read out.

Another possibility is to use Secrets management providers.

complex workflow

This workflow splits the deploying to zenodo in steps. This allows to use the zenodo record (e. g. the DOI) already in the data to publish.

```yaml
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/
```

```yaml
deploy2zenodo:
  rules:
    - if: '"0" == "1"'
      when: never

prepare_deploy2zenodo_step1:
  script:
    - ...

deploy2zenodo-step1:
  variables:
    - DEPLOY2ZENODO_SKIP_PUBLISH: "true"
    - DEPLOY2ZENODO_GET_METADATA: "newmetadata.json"
  extends: .deploy2zenodo
  after_script:
    - echo "DEPLOY2ZENODO_GET_METADATA=$DEPLOY2ZENODO_GET_METADATA" > variables.env
  artifacts:
    paths:
      - $DEPLOY2ZENODO_GET_METADATA
    reports:
      dotenv: variables.env

prepare_release:
  script:
    - echo "use the file \"$DEPLOY2ZENODO_GET_METADATA\""
    - ...

release_job:
  script:
    - ...

prepare_deploy2zenodo_step2:
  script:
    - ...

deploy2zenodo-step2:
  variables:
    - DEPLOY2ZENODO_SKIP_NEW_VERSION: "true"
  extends: .deploy2zenodo
```

In the step `prepare_release` you can use jq to extract data. For example the preserved DOI is available by:

```
jq .metadata.prereserve_doi.doi "$DEPLOY2ZENODO_GET_METADATA"
```

very complex workflow

`deploy2zenodo` uses a combination of the triggered workflow and the complex workflow to publish itself. This is described in deploy_deploy2zenodo_to_zenodo.

script parameter

Instead of command line parameters we use environment variables.

You have to provide the following variables:

variable	content
DEPLOY2ZENODO_API_URL	The URL of the API to use.
DEPLOY2ZENODO_ACCESS_TOKEN	access token of zenodo
DEPLOY2ZENODO_DEPOSITION_ID	id of the deposition/record on zenodo
DEPLOY2ZENODO_JSON	file name with metadata in JSON format to upload
DEPLOY2ZENODO_UPLOAD	file name(s) to upload

There are other optional variables:

variable	content
DEPLOY2ZENODO_SKIP_PUBLISH	prepare record, but skip publishing
DEPLOY2ZENODO_DRYRUN	skip communicating with the external URL
DEPLOY2ZENODO_SKIPRUN	skip everything, only prints commands to execute
DEPLOY2ZENODO_SKIP_NEW_VERSION	skip creating new version
DEPLOY2ZENODO_GET_METADATA	write actual metadata to a file
DEPLOY2ZENODO_SKIP_UPLOAD	skip upload of data
DEPLOY2ZENODO_CURL_MAX_TIME	max time for curl
DEPLOY2ZENODO_CURL_MAX_TIME_PUBLISH	max time for curl during publishing
DEPLOY2ZENODO_ADD_IsCompiledBy_DEPLOY2ZENODO	reference deploy2zenodo
DEPLOY2ZENODO_ADD_IsNewVersionOf	reference previous version
DEPLOY2ZENODO_ADD_IsPartOf	reference DOI for all versions

DEPLOY2ZENODO_API_URL

You can use the API of your own zenodo instance or you can use the official zenodo instance:

state	URL
production	https://zenodo.org/api
testing	https://sandbox.zenodo.org/api

DEPLOY2ZENODO_ACCESS_TOKEN

To access your zenodo account you have to provide an access token.

DEPLOY2ZENODO_DEPOSITION_ID

To update an existing record you have to provide the `id` of this record.

If you want to create a new record please set `DEPLOY2ZENODO_DEPOSITION_ID` to `create NEW record`, e. g. `DEPLOY2ZENODO_DEPOSITION_ID="create NEW record"`. After creating this record read the script output and adapt `DEPLOY2ZENODO_DEPOSITION_ID` for the next run with the returned record `id`.

DEPLOY2ZENODO_JSON

The given file should contain the metadata in JSON format.

You can write this file on your own, e. g.:

```
{
  "metadata": {
    "title": "foo",
    "upload_type": "software",
    "creators": [
      {
        "name": "ich",
        "affiliation": "bar"
      }
    ],
    "description": "foos description"
  }
}
```

You can find the necessary and possible fields on zenodo: Deposit metadata.

Or cffconvert can help harvesting the necessary metadata in JSON format from a CITATION.cff file. Unfortunately we need jq to correct the format, e. g.:

```
cffconvert -i CITATION.cff -f zenodo | \
  jq '{"metadata": .} | .metadata += {"upload_type": "software"}' | \
  tee CITATION.json
```

Since you need to adapt the output of the conversion you can also use more general tools like yq to convert a CITATION.cff file (YAML format) to JSON format.

The JSON format zenodo accepts is much more general and provides many more options than the Citation File Format. For many purposes the CITATION.cff is enough, but otherwise you can see a description of the metadata in the GitHub integration of zenodo[4] [5] [6] using `zenodo.json`, the description of the metadata in zenodo|Developers[7] or InvenioRDM[8] and the unofficial description of zenodo upload metadata schema[9].

As `description` you can use HTML. For example you could use pandoc to convert your `README.md` to HTML and jq to add the HTML code as JSON value (`jq` will escape appropriate characters if necessary):

```
pandoc -o README.html README.md
echo '{"metadata":{"title":"foo","upload_type":"software",
  "creators":[{"name":"ich","affiliation":"bar"}],
  "description":"foos description"}}' | \
  jq --rawfile README README.html '.metadata.description = $README' | \
  tee metadata.json
```

DEPLOY2ZENODO_UPLOAD

The given file(s) will be uploaded as data. Typically this would be an archive.

For example you can create an archive of a tag from a git repository:

```
TAG=0.0.3
git archive --format zip --output $TAG.zip $TAG
```

File names with spaces are not supported. Instead, if `DEPLOY2ZENODO_UPLOAD` contains space(s), the string is split at the spaces. Each individual block represents a file and these files will be uploaded.

[4] developers.zenodo.org GitHub
[5] github.com Referencing and citing content
[6] github: "import" past releases to Zenodo
[7] zenodo|Developers
[8] InvenioRDM: Metadata reference
[9] Zenodo upload metadata schema

The reason not supporting spaces is that you cannot create a CI/CD variable that is an array.

If you really not want to provide data set `DEPLOY2ZENODO_UPLOAD` to `do NOT provide data`, e. g. `DEPLOY2ZENODO_UPLOAD="do NOT provide data"`. If you want to upload 4 files with these names change the order.

Not every zenodo instance supports metadata-only records (configured by `canHaveMetadataOnlyRecords`?). For example the official zenodo instance does not allow metadata-only records! In this case an empty dummy file is uploaded. If this is the case, you should think about respecting the implicit request of the used zenodo instance to provide some data.

DEPLOY2ZENODO_SKIP_PUBLISH

If this variable is not empty the publishing step is skipped, e. g.:

```
DEPLOY2ZENODO_SKIP_PUBLISH="true"
```

Only the record is prepared – metadata and data is uploaded – but not published. You can see what will be published as a preview in the web interface of zenodo and initiate the publishing by pressing the button in the web interface.

This helps to integrate `deploy2zenodo` in your project. But you may also want to curate the upload each time before it is published.

Together with DEPLOY2ZENODO_SKIP_NEW_VERSION this allows to split deploying to zenodo in steps.

DEPLOY2ZENODO_DRYRUN

If this variable is not empty the communication to the given URL is skipped. But your parameters are analyzed. This could help to integrate `deploy2zenodo` in your project.

DEPLOY2ZENODO_SKIPRUN

If this variable is not empty nearly everything is skipped. Only the commands to be executed are echoed. This is for debugging purpose.

DEPLOY2ZENODO_SKIP_NEW_VERSION

If this variable is not empty the step creating a new version is skipped. This allows to split deploying to zenodo in steps.

Between creating a new version and deploying to zenodo you can use the zenodo record (e. g. the DOI) already in the data to publish:

```
jq .metadata.prereserve_doi.doi "$DEPLOY2ZENODO_GET_METADATA" >> README.md
```

Using a manual job allows you to first check the artifacts and data to be published before the last job run.

DEPLOY2ZENODO_GET_METADATA

If this variable is not empty the metadata of the record is stored in a file with this name.

To get these data at the end of the script an additional communication with the DEPLOY2ZENODO_API_URL server is done.

In the CI pipeline you could store the result as artifacts, e. g.:

```
deploy2zenodo:
  variables:
    DEPLOY2ZENODO_GET_METADATA: "result.json"
```

```
  artifacts:
    paths:
      - $DEPLOY2ZENODO_GET_METADATA
```

You can extract values from the metadata. For example to get the DOI to site all versions:

```
deploy2zenodo:
  after_script:
    - jq -r .conceptdoi "$DEPLOY2ZENODO_GET_METADATA"
```

DEPLOY2ZENODO_SKIP_UPLOAD

If this variable is not empty skip uploading the data. This is only allowed if DEPLOY2ZENODO_SKIP_PUBLISH is not empty, too.

If you split deploying to zenodo in steps using DEPLOY2ZENODO_SKIP_PUBLISH and DEPLOY2ZENODO_SKIP_NEW_VERSION you can avoid unnecessary traffic by using also DEPLOY2ZENODO_SKIP_UPLOAD.

DEPLOY2ZENODO_CURL_MAX_TIME

Max time for curl (`--max-time` flag) in seconds for normal use. Default value is 60.

DEPLOY2ZENODO_CURL_MAX_TIME_PUBLISH

Max time for curl (`--max-time` flag) in seconds during publishing. Default value is 300.

DEPLOY2ZENODO_ADD_IsCompiledBy_DEPLOY2ZENODO

If this variable is not empty a reference to deploy2zenodo is added. Something like (but with the DOI of the used version) will be added to your provided JSON file:

```
{
  "metadata": {
    "related_identifiers": [
      {
        "relation": "IsCompiledBy",
        "identifier": "10.5281/zenodo.13871728",
        "scheme": "doi",
        "resource_type": "software"
      }
    ]
  }
}
```

DEPLOY2ZENODO_ADD_IsNewVersionOf

If this variable is not empty a reference to the previous version of your record is referenced. Something like (but with the DOI of the old version and the appropriate resource_type) will be added to your provided JSON file:

```
{
  "metadata": {
    "related_identifiers": [
      {
        "relation": "IsNewVersionOf",
        "identifier": "10.5281/zenodo.10908332",
        "scheme": "doi",
        "resource_type": "software"
```

```
      }
    ]
  }
}
```

This can only work if DEPLOY2ZENODO_SKIP_NEW_VERSION is not used! If you split the run in 2 steps (the first one with DEPLOY2ZENODO_SKIP_PUBLISH and the second one with DEPLOY2ZENODO_SKIP_NEW_VERSION) you have to find the old version in the first run by yourself and provide it in the second run. This is done in deploy_deploy2zenodo_to_zenodo in the jobs `deploy_deploy2zenodo_step1` and `deploy_deploy2zenodo_step2` to publish `deploy2zenodo`. This is something like:

```
step1:
  variables:
    DEPLOY2ZENODO_ADD_IsNewVersionOf: "yes"
  after_script:
    - |
      select(.relation==\"isNewVersionOf\") | .identifier" \
      "$DEPLOY2ZENODO_GET_METADATA")"
    - |
      select(.relation==\"isNewVersionOf\") | .resource_type" \
      "$DEPLOY2ZENODO_GET_METADATA")"
    - |
      {
      echo "LATESTDOI=$LATESTDOI"
      echo "LATESTUPLOADTYPE=$LATESTUPLOADTYPE"
      } | tee variables.env
  artifacts:
    reports:
      dotenv: variables.env

step2:
  needs:
    - job: step1
  variables:
    DEPLOY2ZENODO_SKIP_NEW_VERSION: "true"
  before_script:
    - tmpjson="$(mktemp)"
    - |
      jq ".metadata.related_identifiers += [
      {
      \"relation\":\"IsNewVersionOf\", \"identifier\":\"$LATESTDOI\",
      \"scheme\":\"doi\", \"resource_type\":\"$LATESTUPLOADTYPE\"
      }]" "$DEPLOY2ZENODO_JSON" | tee "$tmpjson"
    - mv "$tmpjson" "$DEPLOY2ZENODO_JSON"
```

DEPLOY2ZENODO_ADD_IsPartOf

If this variable is not empty a reference to all versions of your record is referenced. Something like (but with the DOI of all versions and the appropriate resource_type) will be added to your provided JSON file:

```
{
  "metadata": {
    "related_identifiers": [
      {
        "relation": "IsPartOf",
```

```
        "identifier": "10.5281/zenodo.13871728",
        "scheme": "doi",
        "resource_type": "software"
      }
    ]
  }
}
```

CI pipeline

Using the keyword `include` it is possible to include YAML files and/or CI pipelines in your GitLab CI pipeline. In this way you can use a template of `deploy2zenodo` for your CI pipeline.

You can use the latest version deploy2zenodo.yaml in your CI pipeline. Or you can use any special versions, e. g. deploy2zenodo.yaml v0.1.0.

The provided job is called `deploy2zenodo` and you can overwrite or enhance the defined job as you need (e. g. defining when to run or defining variables).

A simple example choosing the stage to run could be:

```
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/

deploy2zenodo:
  stage: deploy
```

The provided GitLab CI template of `deploy2zenodo` uses `alpine:latest` and installs necessary software curl and jq in `before_script`. To use other images you must adapt it, e. g.:

```
include:
  - remote: 'https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/

deploy2zenodo:
  image:
    name: almalinux:latest
  before_script:
    - echo "nothing to do"
```

script

You can use the script directly. But that is not our focus of `deploy2zenodo`, so we keep it short. For example:

```
SCRIPTURL=https://gitlab.com/deploy2zenodo/deploy2zenodo/-/releases/permalink/latest/downloads/dep
export DEPLOY2ZENODO_API_URL=https://sandbox.zenodo.org/api
export DEPLOY2ZENODO_ACCESS_TOKEN=***
export DEPLOY2ZENODO_DEPOSITION_ID="create NEW record"
export DEPLOY2ZENODO_JSON=metadata.json
export DEPLOY2ZENODO_UPLOAD="foo.zip bar.md"
export DEPLOY2ZENODO_SKIP_PUBLISH="true"
export DEPLOY2ZENODO_DRYRUN=""
export DEPLOY2ZENODO_SKIPRUN=""
export DEPLOY2ZENODO_SKIP_NEW_VERSION=""
export DEPLOY2ZENODO_GET_METADATA="upload.json"
export DEPLOY2ZENODO_SKIP_UPLOAD=""
export DEPLOY2ZENODO_CURL_MAX_TIME=""
```

```
export DEPLOY2ZENODO_CURL_MAX_TIME_PUBLISH=""
curl -L "$SCRIPTURL" | tee deploy2zenodo.sh | sh
```

harvesting

As already mentioned you have to provide the metadata and the data to upload.

In my opinion, this is very dependent on the project. In many programming languages, there is a convention to store metadata such as name, author, description, version and license in certain files (`pyproject.toml`, `library.properties`, ...). In order to deploy this information to zenodo, it must be available in a certain format and with a certain vocabulary. The already mentioned cffconvert tries to do this at least for cff files. Other tools such as somesy have a somewhat different focus, but they can also help in a pipeline/toolchain. For example, you could use it to convert a typical python `pyproject.toml` into `CITATION.cff` and then use cffconvert and jq to get metadata for zenodo. However, hermes should also be mentioned here. hermes tries to merge metadata from different sources and to provide it for zenodo.

curating

You have various options for curating the data for publication.

The typical workflow in software development is to work in developer or feature branches and then merge them with the default branch (e. g. `main`). This is usually done in a merge request. If the harvesting of metadata and data is already taking place in a CI pipeline at this point, this can also be checked in the merge request.

If publishing is prevented by using `DEPLOY2ZENODO_SKIP_PUBLISH`, the preview in the zenodo web interface can be used to check the result.

If you have implemented a stable, functioning process, curation can also be omitted and publishing can be fully automated.

license: Apache-2.0

`deploy2zenodo` has the license Apache-2.0.

```
Copyright 2023, 2024 Daniel Mohr and
   Deutsches Zentrum fuer Luft- und Raumfahrt e. V., D-51170 Koeln

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```