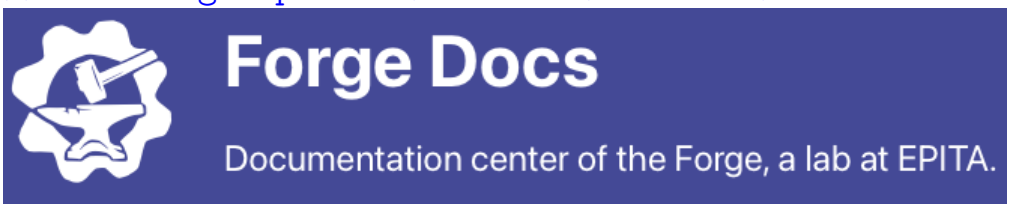
Forge Flow

Jim Newton

September 22, 2026

1 Introduction

This document describes the process for creating and maintaining projects using the EPITA Forge/Intranet service: https://intra.forge.epita.fr. The Forge Docs https://docs.forge.epita.fr also refer to this as *Project Hosting (PaCv2)*. Documentation for the service is available here: https://docs.forge.epita.fr/services/intranet/introduction.



This document is a compilation of best-known practices that we have developed over several years of usage. DISCLAIMER: Many of the recommended practices described herein may possibly be out of date, as it is altogether possible that we are working around bugs which no longer are bugs in the system. If the reader discovers such a case, we welcome feedback to improve and simplify our flow.

The Forge/Intranet service framework provides a robust environment for assigning programming assignments to EPITA students (Engineering students, International Bachelor, Cyber-Bachelor, and Apprentissage students). However, setting up a project is *difficult*, and a great many things can go wrong.

Despite the difficulties we have encountered, we find the system far better than the previous method which was that each and every professor was developing an ad hoc system which involved running student insecure code on the professor's laptop.

Contents

2 Overview

In Figure 1, the repositories in blue are initially created (each year/semester) by the Forge/Intranet service, either empty or seeded, and must be maintained by the professor to implement the particular course assignments desired. The repositories in pink are created once and span the several-year life cycle of the EPITA course. Periodically a `Makefile` target must be run to copy data from the professor repository to one of the other repositories, or to update or test the moulinette repository.

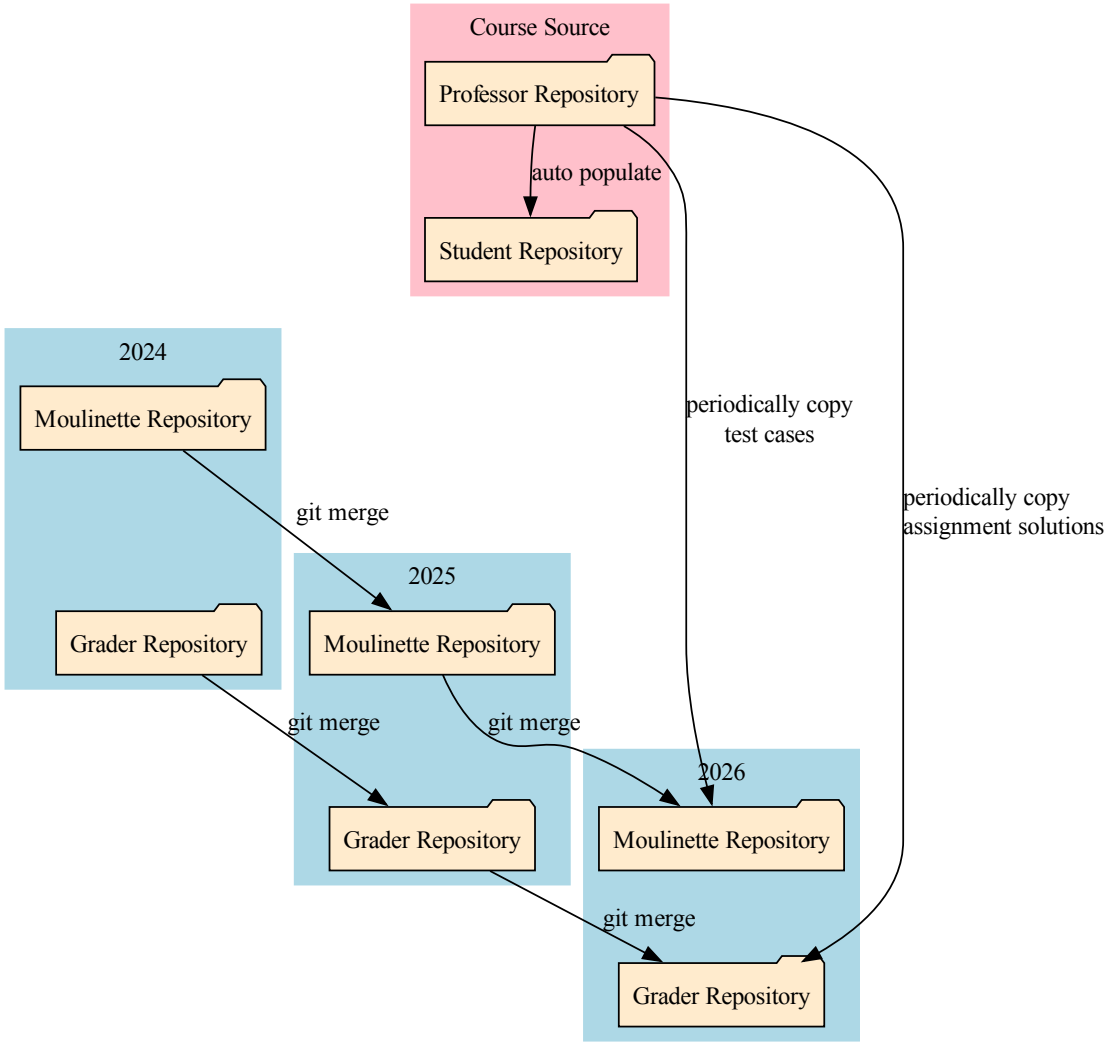


Figure 1: Overview of Repositories

2.1 Repositories

There are several git repositories which are used in a Forge project.

- The *moulinette* repository, Section 3.
- The *grader* repository, Section 4.
- The *professor* repository, Section 5.
- The *student* repository, Section 6.

Distinct from these four, one more repository, `maas-courses` (Section 2.2), holds personal, cross-course bookkeeping that Forge itself has no notion of — which git URLs belong to which course-year, and the grade-computation working directories.

2.2 The `maas-courses` Repository

`maas-courses` (`git@gitlab.cri.epita.fr:jim.newton/maas-courses.git`, cloned locally at `~/Repos/courses/maas-courses`) is a personal, manually-maintained repository, not part of the Forge/Intranet-managed four-repo model above. It documents nothing about the general MaaS/Forge scheme — that stays in this document, forge-flow — only concrete data about this user's actual courses. It holds two things:

- `registry.json` and `bin/registry.csh` (Section 2.2.1) — a lookup table of every course-year's four repository URLs (and a handful of other per-course-year facts), plus the one controlled script that reads and writes it.
- `courses/` — the grade-computation working directories referenced in Section 3.5: one subdirectory per course-year, holding `compute_grades.clj`, `announce_grades.py`, `grades.csv`, exam rosters, and the shared `*_incl.{clj,py}` files they `load-file`/import. Migrated out of `~/Documents/courses` (history preserved via `git filter-repo`); that older location is retired in favor of this repository.

2.2.1 File `registry.json` and Script `bin/registry.csh`

`registry.json` records, for every course the user has taught (keyed by course slug, then by year), the git URLs of that course-year's Professor/Student/-Moulinette/Grader repositories, their default branches, the school's own internal course code (for cross-referencing the official timetable), and which exact commit of the Professor Repository fed the Moulinette repository's `docker/suite` at the last `copy-tests.csh` sync. The file's own top-level `"_schema"` key documents every field's meaning in full — JSON has no comment syntax, so the documentation lives there instead of here, and is the authoritative reference, not this paragraph.

`registry.json` should never be edited directly, and never `jq`'d inline from a caller script. `bin/registry.csh` is the single controlled interface, callable by absolute path from any repo, in any language (csh via backticks, Python via `subprocess`, Clojure via `clojure.java.shell/sh`):

Listing 2.1 (*`registry.csh` usage*)

```
1 registry.csh get <course> <year> <field>
2 registry.csh set <course> <year> <field> <value>
```

`get` prints the field's value, or nothing (exit 0) if unset. `set` bootstraps `registry.json` if it doesn't exist yet, updates only the named field (leaving every other field, and the `"_schema"` key, untouched), then commits and pushes — so the jq/bootstrap/commit-and-push logic lives in exactly one place instead of being duplicated inline in every caller script. *E.g.*, `copy-tests.csh` (Section 3.4.9) calls `registry.csh set` to record the Professor Repository's commit after every `docker/suite` sync.

2.3 Requirements

We use several UNIX tools in this flow which must be installed on your system for the flow to work.

- `csh`, many of the scripts are written using `csh`.
- `clojure`, many of the scripts are written in `clj` which is the clojure CLI scripting environment.
- `forge`, this is the CLI (command-line-interface) to talk to the Forge/Intranet server. The application must be installed on your machine. It can

be installed using the `pip` command.

Listing 2.2

```
pip install \
  'git+ssh://git@gitlab.cri.epita.fr/forge/tools/cli'
```

The `forge` program needs you to authenticate using the following:

```
forge auth token
```

You'll be requested to open a web page and authenticate. This saves the authentication information in a cache file. The cache file sometimes gets corrupted if you stop a program while it is running, or if you run `forge` twice in different threads or different shells. To clear up the corruption, do the following

Listing 2.3

```
rm ~/.config/forge-cli-credentials.json
forge auth token
```

If you are unable to install using `pip install` you may have to use `pipx install` to install local versions.

- `glab`, the official GitLab CLI. Some scripts poll the GitLab REST API directly – *e.g.* to wait for a `docker-build` CI pipeline to finish and confirm the resulting image tag landed in the container registry, rather than checking the pipelines and container-registry web pages by hand after every `make docker-image`. Install it with `brew`:

  Listing 2.4

  ```
  brew install glab
  ```

  Then authenticate:

  Listing 2.5

  ```
  glab auth login --hostname gitlab.cri.epita.fr
  ```

  `glab auth login` defaults to authenticating against `gitlab.com`, not the self-hosted instance this flow actually uses. Always pass

`-hostname gitlab.cri.epita.fr` explicitly, or `glab auth status` will report a working configuration for the wrong host while every API call against the real project still fails with 401 Unauthorized.

This authentication is per-machine and per-host, *not* per-repo or per-course: a single `glab auth login` on a given laptop covers every Moulinette/Grader repository on `gitlab.cri.epita.fr`, present and future, since the underlying token is scoped to your account on that GitLab instance, not to an individual project. It only needs to be repeated when setting up a new laptop, or if the token expires or is revoked – never when creating a new course's Moulinette repository. Scripts that use `glab` should check `glab auth status -hostname gitlab.cri.epita.fr` before doing real work and print setup instructions on failure rather than fail with an opaque error – see `wait-for-docker-image.csh` in `bcs-calculus-l3-2026` for an example.

During `glab auth login` you will be walked through a few more prompts:

- *Container registry and image dependency proxy domains* – accept the suggested default (it derives correctly from the hostname).
- *Token, Web, or Device?* – choose *Token*. *Web* requires an OAuth application to be registered on the GitLab instance (a `client_id`), which is not currently set up for `gitlab.cri.epita.fr`; choosing it fails with `Set 'client_id' first with ...`. Token instead needs a Personal Access Token, created at https://gitlab.cri.epita.fr/-/user_settings/personal_access_tokens with the `read_api` scope only (sufficient for reading pipeline status and the container registry; no write access is needed).
- *Git protocol?* – choose *SSH*, matching how every repository in this flow is already cloned.
- *Host API protocol?* – choose *HTTPS* (never HTTP – it would send the token in cleartext).

- The `forge` program needs the Python `certifi` library to be installed. Install it with `pip`.

Listing 2.6

```
1 pip install certifi
```

- There are MacOS dependencies. In particular many scripts use a program named `trash` which is like `rm` except that it moves the file to the `Trash` and succeeds even if the file is in use. The `trash` program also robustly handles renaming if necessary, *e.g.*, if there is already a file of the same name in the `Trash`.

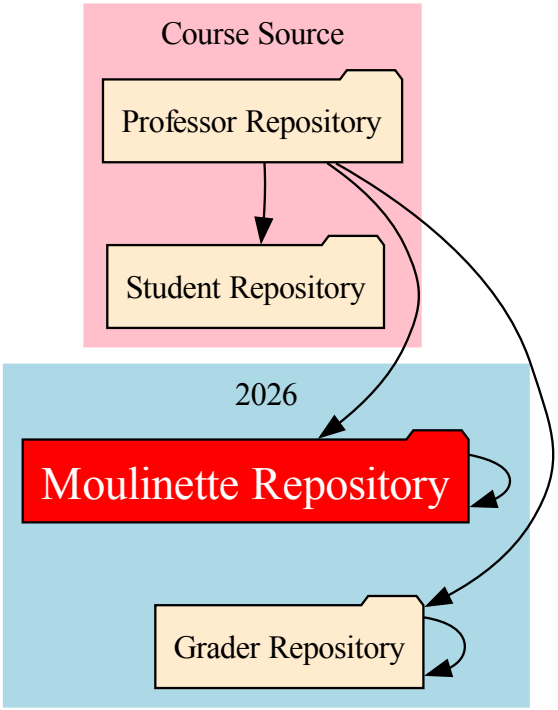


Figure 2: Moulinette Repository

3 Moulinette Repository

The Forge team refers to their auto-grader engine as MaaS (Moulinette as a Service) — the cloud successor to the traditional EPITA *moulinette*, the local auto-grader script that professors previously ran on their own machines. Throughout this document we call it the *auto-grader*.

This repository specifies:

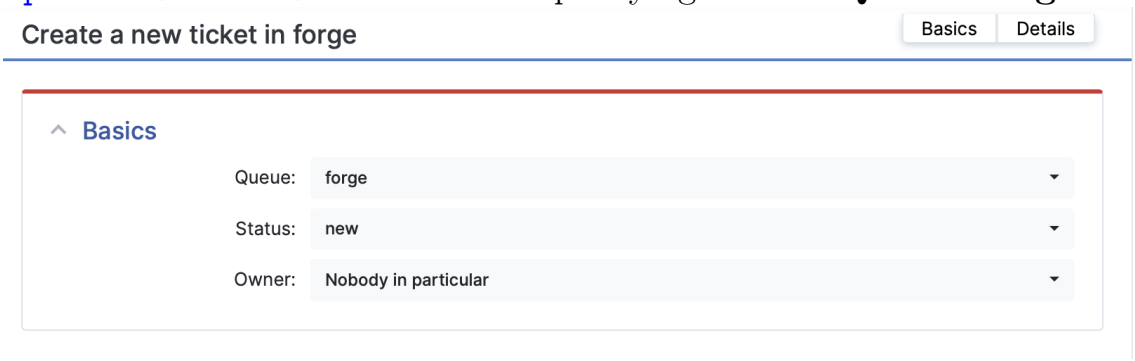
- The content of the student assignments,
- The student-facing instructions (subjects [fr]) of student assignments,
- The dates (opening, closing, etc) of the student assignments,
- The content of the Docker image,
- The auto-grading scripts to evaluate student submissions,
- The list of students for the course.

The Forge/Intranet service expects commits and tags to be found on the `main` branch (not `master`) of this repository. Commits on the `master` branch are a source of difficult to find errors in the flow. The grader repository (Section 4) is expected to have commits and tags on the `master` branch (not `main`). This `master`/`main` is a potential source of confusion.

3.1 Creating the repository

You cannot create this repository; CRI must do it for you and give you the URL.

To establish a new course, the CRI group must create a new moulinette repository. The professor must submit a ticket using https://tickets.forge.epita.fr/Ticket/Create.html specifying **Basics Queue: forge**.



Once the ticket is completed by the CRI group you will be given a URL for a repository such as git@gitlab.cri.epita.fr:ing/activites/2025-cloj-elec-promo-2027.git, a tenant such as `epita-pi-cs-bachelor`, `epita-apprentissage`, or `epita-ing`, and finally a token such as `7b4281b8-cf94-4afa-99f8-e644adfddde4`. You'll use the token to create your `ACTIVITY_TOKEN` in the gitlab configuration as explained here: https://docs.forge.epita.fr/services/intranet/activities/.

3.2 Setting up a course from a previous year

If you want this year's course to use the same (or similar) assignments as a previous year, you must ask CRI to create the Repository for you as in Section 3.1. This will give you a repository with many files which you probably won't understand.

You'll need to register the `ACTIVITY_TOKEN` token as explained in Section 3.1.

Thereafter, you can import the previous year's files into this repository. This can probably be done with a copy, but we recommend using `git remote`.

Suppose the moulinette repository of the new course is named `2026-xyz` and the previous course (moulinette repository) is named `2025-xyz`.

Listing 3.1 (*Merge in Last Year's Repository*)

```
# csh syntax
set newyear 2026-xyz
set oldyear 2025-xyz
cd $newyear
git checkout -b main

# Since we cannot merge onto an empty branch, we need
#   to create a dummy initial commit.
date > erase.me
git add erase.me
git commit -mInitial-commit
git rm erase.me
git commit -m"removed useless file"

# Now add last year's repo as remote and merge it
#    onto main
git remote add $oldyear \
      git@gitlab.cri.epita.fr:path/to/$oldyear.git
git fetch $oldyear
git tag -d `git tag`
git merge --no-edit --allow-unrelated-histories \
      main $oldyear/main
git remote remove $oldyear

# remove last years students to prevent repositories
#    from being created for the wrong students
echo > people.yml
git add people.yml
git commit -m"removed last year students"
git push
```

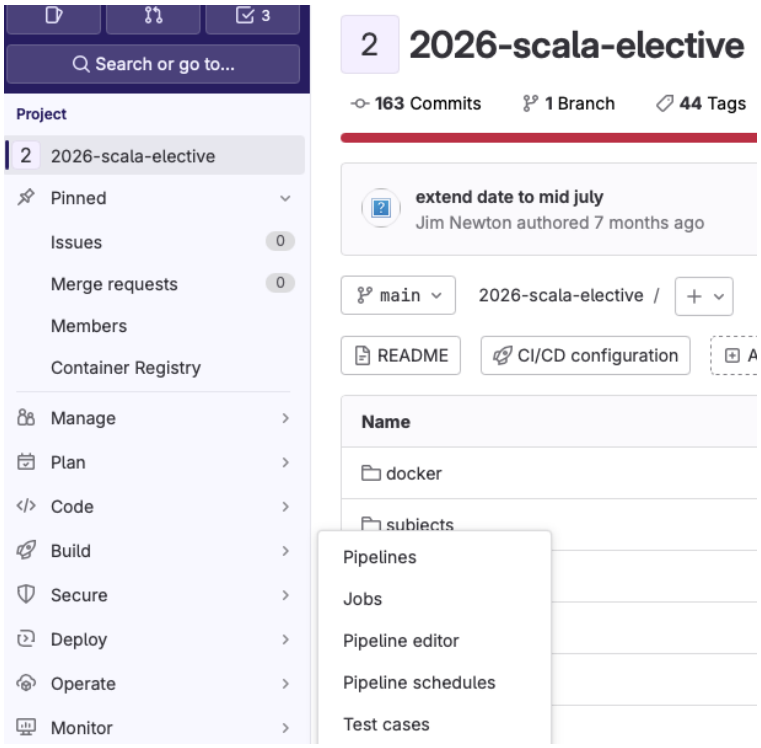
You'll probably need to change the following things.

- Do not forget to remove last year's students from the `people.yml` file (Section 3.4.6). If you leave last year's students in the file, those students will be added to this year's class, and the students will probably contact you asking why they have to re-take the class.

- Now you'll have to find all references to `2025-xyz` and change to

`2026-xyz`. This includes all references to the old course name in `activity.py` (Section 3.4.3), `copy-tests.csh` (Section 3.4.9), `test-assignments.csh` (Section 3.4.8), and all the `.html` files in the `subjects/` directory (Section 3.4.12).

- The assignment dates in the `activity.py` (Section 3.4.3).
- Visit all the `.yml` files in `workflows/`, and update the `image:` tag (Section 3.4.13).
- Run `make activities` and `make docker-image`. You may monitor the build from GitLab. Find the **Build**/**Pipelines** menu item.



You will see which pipelines failed, and you can examine the log to find

and fix the errors.

- Once `make activities` runs successfully, a empty grader repository (Section 4) will have been created for you which you can clone. You may populate this repository simply with a copy (from the previous year) or using `git remote`, and `git merge -allow-unrelated-histories` similar to how you populated the moulinette repository.

3.3 Setting up a new course from scratch

The process of creating a course from scratch is of course more involved than copying one from the previous year.

One of the important things you'll have to set up is a mechanism for grading a student's submission. The flow for this depends on the programming language the students are using. We have models for this already for several languages `Scala`, `Python`, and `Clojure`. From a high level, you'll need a script to run from the shell which will launch a grader which leaves a single `/output.xml` file at the top level.

- **Scala:**

```
sbt -Dsbt.log.noformat=true compile "testOnly $suite"
```

With the following lines in the `build.sbt` file.

Listing 3.2 (*Insert into `build.sbt` file*)

```
1  Test/testOptions += Tests.Argument(TestFrameworks.ScalaTest,
2                                     "-u",
3                                     "scalatest-output")
```

- **Python:**

  Listing 3.3

```
python3 -m pytest tests/*public_test.py \
    tests/*private_test.py -v --junitxml="$xmlfile"
```

- **Clojure:**

  Listing 3.4

```
lein test homework.${suite}-test >& clojure_test.out
```

  With the following lines in the `defproject` in the `project.clj` file.

  Listing 3.5 (*Insert into `project.clj` File*)

```
:profiles
   {:test {:plugins [[lein-test-report-junit-xml "0.2.0"]]
           :test-report-junit-xml {:output-dir "."}}
    :uberjar {:aot :all
              :jvm-opts
                ["-Dclojure.compiler.direct-linking=true"]
             }}
```

The format of this `.xml` is specified by a Java/ `JUnit` specification, but all languages we've investigated have a way to produce this file. Unfortunately, languages usually don't format the file exactly like Forge is expecting it. Therefore, you must pass the XML output through the script `fix-xml.py` (Section 3.4.20) before creating the `/output.xml` file. This script finds many common discrepancies and repairs them in a way that the student-facing web page can display meaningful error messages.

The current implementation of `docker-entrypoint.sh` (Section 3.4.17) handles extracting the student's submitted code and running `launch_testsuite.csh` (Section 3.4.18) with a single command line argument which comes from the `command:` keyword in the corresponding `.yml` file in the `workflows/` directory (Section 3.4.13).

3.4 Files in Moulinette Repository

This section contains a detailed list of all the important files in the Moulinette repository.

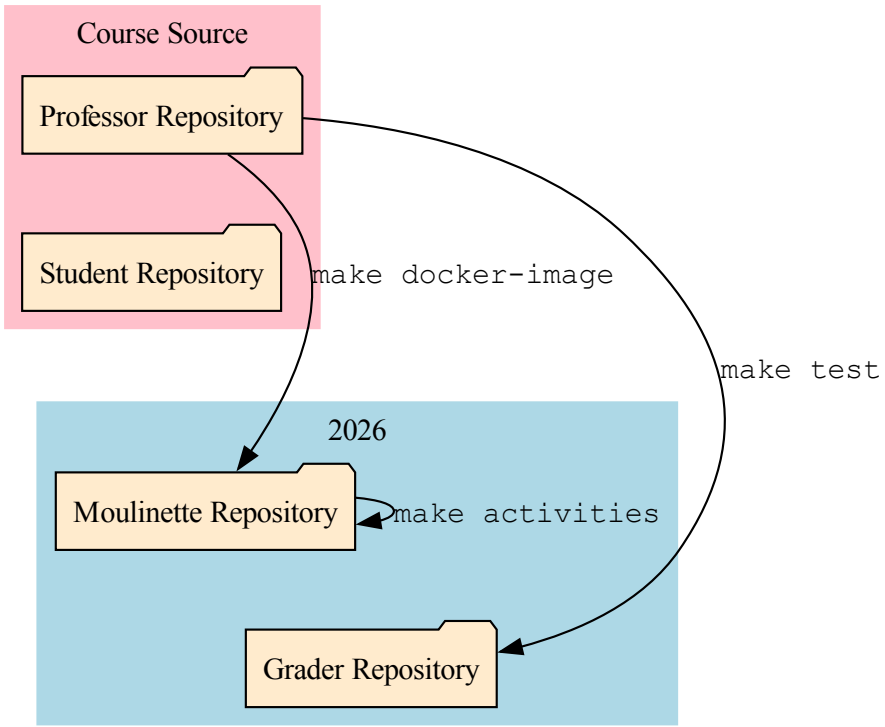


Figure 3: Moulinette Repository `Makefile` Targets: `activities`, `docker-image`, `test`

3.4.1 File `Makefile`

Figure 3 illustrates the action of the `Makefile` targets in the moulinette repository.

- Make target: `activities` to regenerate the `activity.yml` from the `activity.py` file, and `git push`. This updates the student view of the assignments on the Forge web page. *E.g.*, the dates of the assignments *etc.* Pushing to `main` triggers a validate+deploy CI pipeline (the deploy stage is what actually syncs `activity.yml` / `activity.json` to Forge/Intranet); `wait-for-activities-ci.csh` polls that pipeline (matching by commit sha, since `main` is pushed to directly rather than tagged) so the target doesn't return before the sync has actually happened. `sync-activity-json-to-student-repo.csh` then pushes `activity.json` on to the Student Repository, a second copy independent of the Forge/Intranet sync above – Section 3.4.4 explains why that copy exists and is pushed automatically here.

Listing 3.6 (*Make target* `activities`)

```
activities: activity.yml activity.json people.yml
	date > .gitkeep
	git add .gitkeep
	git add --ignore-errors activity.*
	git add --ignore-errors *.yml
	git commit -m'updated activity.yml'
	git push
	./wait-for-activities-ci.csh
	./sync-activity-json-to-student-repo.csh
```

A professor-only self-test submission (*e.g.* `Prof-{slug}` in `activity.py`) that reuses `courseBegin` for its own `publishingAt` / `openingAt` will silently stop accepting pushes to the Grader Repository once `courseBegin` is corrected to the real (future) semester start date – the symptom is a git push failing with `Cannot access to repository ...` even though the same repository clones (reads) fine and `glab`/SSH authentication itself succeeds. Give professor-only submissions their own always-past date (*e.g.* a `profTestingBegin` constant), independent of `courseBegin` / `courseEnd`, so the professor's own pipeline testing is never accidentally gated by the real course schedule. Diagnosed and confirmed fixed in `bcs-calculus-l3-2026` on 2026-08-21.

- Make target: `docker-image` to make a new docker image with a new version number. This target is necessary if something has changed related to running the tests in the Docker image. *E.g.*, if the test cases have changed, or if new tests were added.

Listing 3.7 (*Make target* `docker-image`)

```
1 docker-image: activities
2       ./sync-docker-harness.csh
3       ./copy-tests.csh
4       NEWVER=`./suggest-next-version.csh` ; \
5         ./new-version.csh $$NEWVER ; \
6         ./wait-for-docker-image.csh $$NEWVER
```

`sync-docker-harness.csh` refreshes `docker/` from the canonical, byte-identical-across-courses copies in `forge-flow/scripts/docker/` – see Section 3.4.15. `wait-for-docker-image.csh` polls the CI pipeline triggered by `new-version.csh`'s tag push and confirms the resulting image landed in the container registry, so the target doesn't return before the new image actually exists.

- Make target: `test` runs a script named `test-assignments.csh`. See Section 3.4.8 for details.

Listing 3.8 (*Make target* `test`)

```
1 test:
2       ./test-assignments.csh
```

- Make target: `grades` runs a script named `compute_grades.clj` This target is deprecated; grade computation is now handled by per-course scripts outside the moulinette repository. See Section 3.5.

3.4.2 File `activity.yml`

This file is the starting point of a two-step generation chain triggered by `make activities`. Do not edit it by hand; edit `activity.py` instead and re-run `make activities`.

The full chain is:

1. `python activity.py` ⟶ `activity.yml`

The `activity.yml` file is read by the Forge/Intranet service to define assignments, tag prefixes, dates, and student subscriptions for the course.

2. `python yml-to-json.py` ⟶ `activity.json`

   A JSON representation of the same data. This file is consumed at runtime by the grade-computation library (Section 3.5), which clones the moulinette repository in order to read assignment dates and the student list. It is also read directly (via `jq`) by `tag-prof.csh`[1] to find the `Prof-*` tag prefixes to push when self-testing the grading pipeline. A third consumer, `submit-to-grader.py` (Section 8.3), reads a *different* copy of this same file entirely – Section 3.4.4 explains why.

Both generated files (`activity.yml`, `activity.json`) should be committed to the moulinette repository after each run of `make activities`.

`activity.json`'s Makefile rule must list every file `activity.yml` pulls in via a literal `!include` directive (`people.yml`, or per-group files such as `L2.yml` / `L3.yml` or `g1.yml` / `g2.yml` / `g3.yml`) as prerequisites, not just `activity.yml` itself:

Listing 3.9 (*Correct `activity.json` rule (calculus example)*)

```
activity.json : activity.yml L2.yml L3.yml
	${PYTHON} yml-to-json.py
```

`activity.yml` itself never changes when a student is added – it only contains the literal string `!include L3.yml`, not resolved logins – so `make` has no other way to know `activity.json` depends on `L3.yml`'s content too: `yml-to-json.py`'s custom YAML loader resolves `!include` by reading that file only at the moment it runs, invisibly to `make`'s own dependency tracking. Get this wrong (as every one of `bcs-calculus-l3-2026`, `2025-algebra-1`, `2026-linear-algebra`, and `2026-ads` did until 2026-09-01) and adding a student to the roster file silently produces no new `activity.json` at all: `make activities` reports success, commits nothing new, and the student is invisible to `submit-to-grader.py` with no error anywhere. Check for this bug in any older course by touching the roster file and running `make activity.json` – it should rebuild.

[1] In the Grader Repository – see Section 4.

An earlier revision of this flow also generated `assignment-tags.txt` (via `clojure extract_tags.clj`), a plain-text list of tag prefixes duplicating what `activity.json` already contains. It was removed 2026-08-21: its only actual consumer was `tag-prof.csh`, switched to reading `activity.json` directly instead, and removing the intermediate file eliminates the risk of it going stale relative to `activity.json` between runs. If you find an older course whose moulinette repository still generates or commits this file, it is safe to drop both the file and the `extract_tags.clj` / `assignment-tags.txt` Makefile rule, matching `bcs-calculus-l3-2026`.

3.4.3 File `activity.py`

The command `python activity.py` recreates the `activity.yml` file. The `activity.py` may be modified to adjust details about assignments such as opening and closing dates. The `activity.py` also associates each assignment with the tag prefix which the student must use to submit the assignment to the auto-grader.

We use three different sets of tag prefixes, one for each assignment.

- The normal tag: *e.g.*, `git tag Hello-2025-04-01-18-12`.

  The normal tag such as `Hello` intended for the student to use to submit an exercise. This tag is usually only valid for a limited period of time so that the assignment is invisible to the student before and after a certain period. The time period is configured in the `activity.py`. This limited visibility makes it difficult to test, because the professor is also prohibited from tagging with this tag, which prevents testing.

- The `Prof-` prefix tag: *e.g.*, `git tag Prof-Hello-2025-04-01-18-12`.

  We configure the `activity.py` file to also generate a `Prof-` tag prefix such as `Prof-Hello`, for each exercise. This tag prefix is only visible to the professor but is visible over the entire lifetime of the course. This lifetime allows the professor to test the activity. To test the activity use either `make -f Makefile-tag prof` from within the grader repository, or `make test` within the moulinette repository.

- The `Late-` prefix tag: `git tag Late-Hello-2025-04-01-18-12`.

  We configure the `activity.py` file so that at the moment the normal tag goes out of visibility, another tag with the `Late-` prefix comes into

visibility, and remains visible either for another week or for the entirety of the course. If a student failed to submit a homework assignment on time, the student can still submit late for a penalty. We usually apply a 40% penalty to late submissions, however the professor is free to waive this penalty if special circumstances exist, such as network problems, or if there were errors in the test suite itself which caused confusion to the students.

3.4.4 File `sync-activity-json-to-student-repo.csh`

`submit-to-grader.py`[2] decides which submissions are currently open by fetching `activity.json` itself, live, via a fresh `git clone -depth=1` of the Student Repository, every time the student runs it – never a locally cached copy, so that today's date always governs which submissions show as open, and never the copy in the moulinette repository, which a student has no access to. That means the moulinette repository's own `activity.*` being correct and pushed (the `activities` target's first `git push`, synced to Forge/Intranet by CI) is *not enough* on its own – a second, independent copy of `activity.json` has to reach the top level of the Student Repository too, or `submit-to-grader.py` keeps reading whatever was there before.

`sync-activity-json-to-student-repo.csh` is that second push: it resolves the Student Repository URL via `registry.csh` (the `maas-courses` interface – Section 2.2), clones it fresh, copies the local `activity.json` to its top level, and commits and pushes only if that changed anything. It is wired in as the last step of the `activities` Makefile target (Section 3.4.1), after `wait-for-activities-ci.csh`, so it runs unattended every time and reflects the CI-validated copy.

Deliberately narrow, unlike a course's own `update-student-repo.csh` (Professor Repository, not moulinette – republishes the whole `doc`/`src` course-materials tree and stays a deliberate, professor-run publish step, see Section 5): this script touches nothing in the Student Repository but `activity.json`, which is exactly why it is safe to run automatically on every `make activities` rather than needing a human to remember a separate step. Folding a full materials publish into an unattended roster-sync step would risk pushing whatever half-finished state `doc`/`src` happen to be in at that moment – keeping the two separate keeps that judgment call with the professor.

Like `wait-for-activities-ci.csh`, this script is not one

[2] The newer version, in use for `calculus` and `algebra-1` as of 2026-09-01 – see Section 8.3.

of the byte-identical files `sync-docker-harness.csh` pulls into `docker/` on every build (Section 3.4.15) – it needs a per-course `registry_course` / `registry_year` pair, which cannot itself be generic. Instead, when bootstrapping a new course, copy it byte-identical from `forge-flow/scripts/sync-activity-json-to-student-repo.csh` and edit only its CONFIGURATION block:

Listing 3.10 (*CONFIGURATION block, edited per course*)

```
1 set registry_course = calculus
2 set registry_year = 2026
```

3.4.5 File `.gitlab-ci.yml`

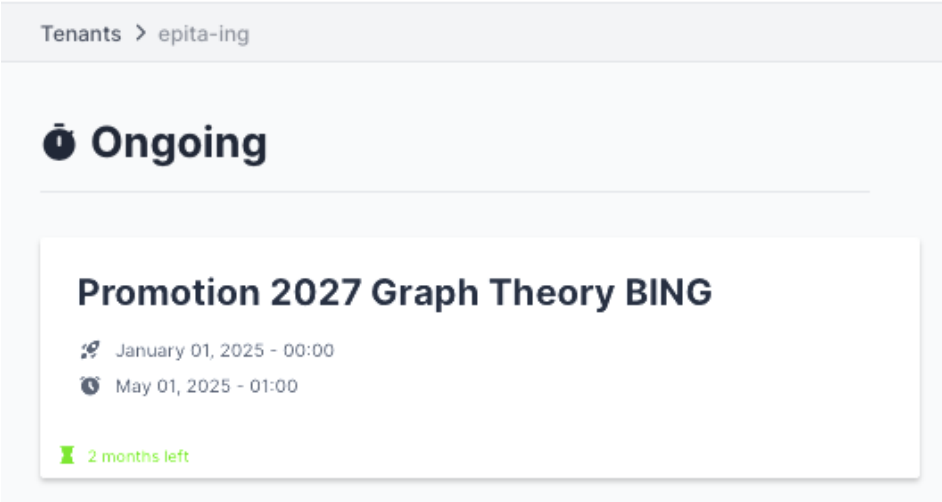
I've never had an occasion to modify this file. It came this way, and I left it as-is. According to https://spacelift.io/blog/gitlab-ci-yml, `.gitlab-ci.yml` is the main YAML configuration file for GitLab CI/CD. GitLab projects that use CI/CD features have a `.gitlab-ci.yml` file located in their repository's root directory. GitLab detects the file on pushes and merges, then parses it to discover the pipeline jobs to run. Created jobs are executed by an available GitLab Runner instance.

3.4.6 File `people.yml`

This file lists the students registered for the course. The forge system will assure that each student has an individual grader repository available (initially empty). That grader repository will be used by the student to submit assignments. The name of the professor should also be listed as a student in this file, otherwise the professor will not be able to test submissions.

A student's name in this file will cause the course to become visible on the

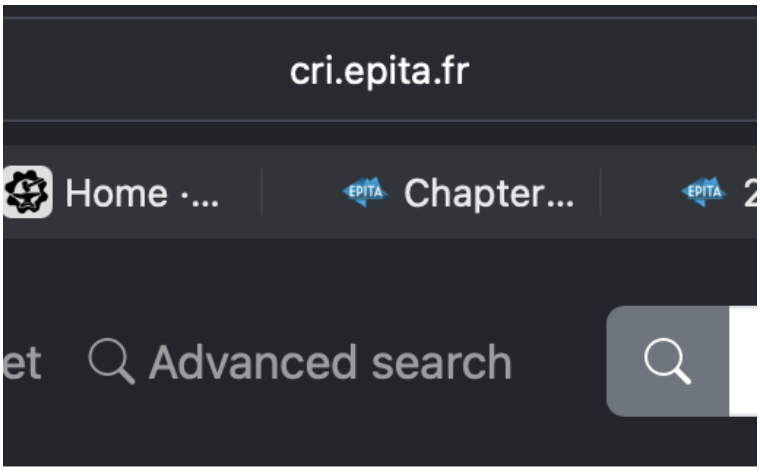
student's Forge web page, https://intra.forge.epita.fr, such as:



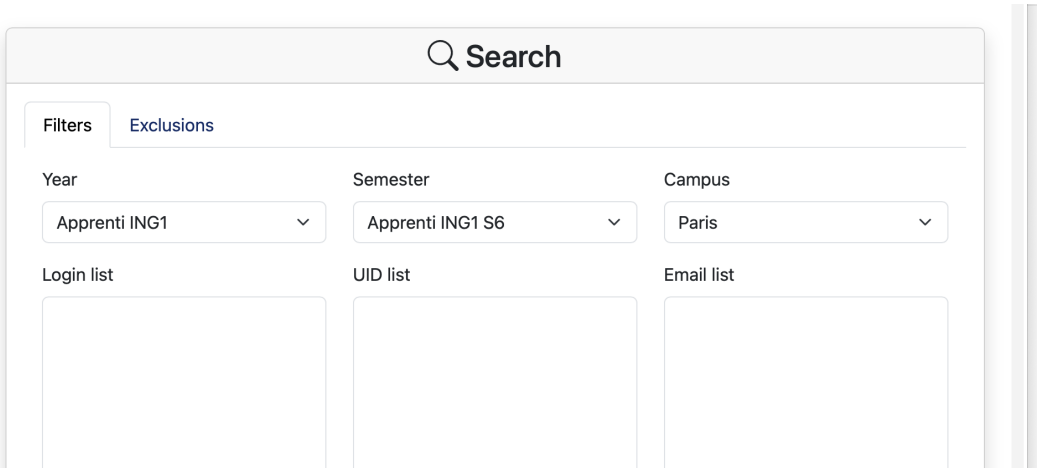
If a student complains that the course does not appear on https://intra.forge.epita.fr, then make sure the student's name is in this file and is spelled correctly. The student's name should have the same spelling as the `@epita.fr` login, the username for https://gitlab.cri.epita.fr, and the profile for https://cri.epita.fr.

If you create this year's course by copying or pulling from last year's course, you'll get last year's `people.yml` file.

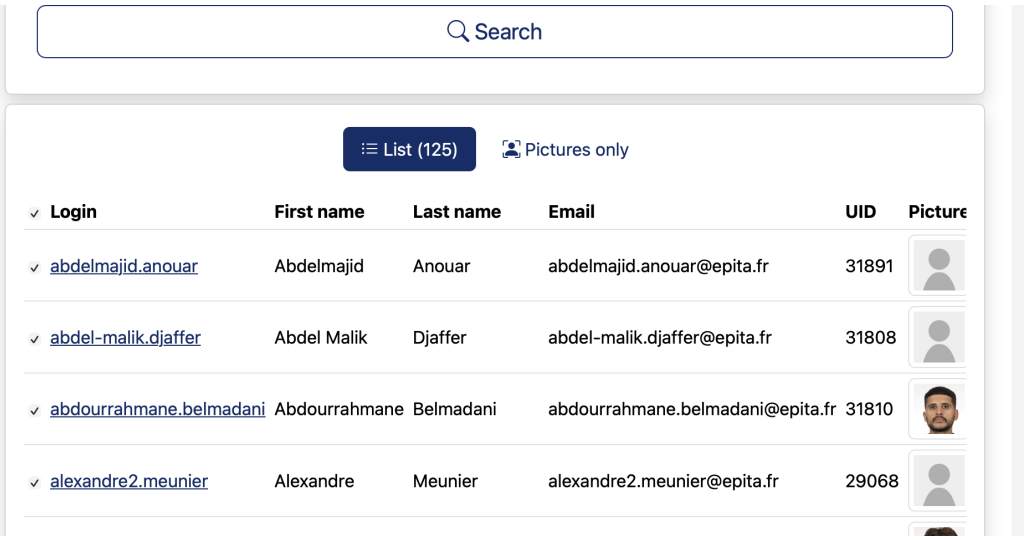
If all students in a particular promotion (graduation year) are enrolled in the course, you can find the list of email addresses using the **Advanced search** facility on the cri.epita.fr web page.



Enter the promotion year, semester, and campus.



Press **Search**, and scroll down the page.



Unfortunately, there is no *export-to-csv* feature on this page. You'll simply need to copy with the mouse, and paste into your favorite editor to construct the `people.yml` file.

The format of the file looks like this, usually putting the professor's name at the top of the file.

Listing 3.11 (*Content of `people.yml` file*)

```
- jim.newton
- abdelmajid.anouar
- abdel-malik.djaffer
- abdourrahmane.belmadani
- alexandre2.meunier
- alexandre.bakiri
- alexis.burnaz
```

3.4.7 File `compute_grades.clj`

This file is deprecated. Grade computation is now handled by per-course scripts that live outside the moulinette repository. See Section 3.5 for the current approach. This file may be safely removed from the moulinette repository.

3.4.8 File `test-assignments.csh`

This script checks out the grader repository (Section 4) in `/tmp` and runs `make -f Makefile-tag copy` and `make -f Makefile-tag prof`.

3.4.9 File `copy-tests.csh`

Only the CONFIGURATION block (Professor Repository URL, registry course/year, and the `sync_specs` declaration) is course-specific; the generic logic is synced from a canonical copy – see Section 3.4.15.

This is a shell script run by the make target `copy`. The script copies all necessary code into the hierarchy under the `docker/` directory to allow the auto-grader to safely and securely run the student submission.

It is the responsibility of `copy-tests.csh` to populate the directory `docker/suite` (Section 3.4.21).

3.4.10 File `new-version.csh`

This script updates the version number of the repository, registers it in all the `.yml` files in the `workflows/` directory, creates a git tag, and pushes it. Typically, `new-version.csh` is run with a single command line argument `‘suggest-next-version.csh‘`, but may be run manually with another argument in the case of debugging.

3.4.11 File `suggest-next-version.csh`

This script must be modified when setting up the moulinette repository. The script uses the command `git rev-parse -show-toplevel` to determine the name of the repository.

3.4.12 Directory `subjects/`

In some cases there are `.html` files in the `subjects/` directory, one file per assignment. These files give human-readable instructions to the student of how

to complete the assignment in question.

If you wish to have individual descriptions per exercise, you'll need one `.html` per exercise and that `.html` should be referenced from the `activity.yml` (which is auto-generated from `activity.py`). The following example YAML code specifies such a subject file `subjects/SimpleGraphs.html`, in the `documents:` section.

Listing 3.12 (*Linking `activity.yml` to a `.html` file in `subject/`*)

```
  - slug: SimpleGraphs
    name: "Exercise 1: Simple Graphs"
    maxLearners: 1
    subscriptions:
      logins: !include people.yml
    documents:
      - path: workflows/SimpleGraphs.yml
        name: maasWorkflow.yml
        filename: maasWorkflow.yml
        type: WORKFLOW
        visibility: PRIVATE
      - path: subjects/SimpleGraphs.html
        name: SimpleGraphs.html
        filename: SimpleGraphs.html
        type: EMBEDDED
    submissions:
        ... STUFF OMITTED ....
```

3.4.13 Directory `workflows/`

This directory contains `.yml` files, one per assignment. The content of each file specifies a keyword `command:` which is data used by `docker/launch_testsuite.csh`, where it is referred to by the variable name `suite`. In this file, you may specify a non-default time-out or amount of memory for auto-grading the assignment. Here is an example of such file content.

Listing 3.13 (*Assignment Declaration in* `workflows/`)

```
cpu_count: 1
memory_mb: 1024
timeout: 300
stages:
 - name: moulinette
   tasks:
   - name: testsuite
     cpus: 1
     memory_mb: 1024
     timeout: 300
     input_file_path: /maas-bundle.tar
     output_file_path: /output.xml
     image: "registry.cri.epita.fr
        /ing/activites/2025-theg:v1.1.1"
     # prefix of testing files
     command: ["gt"]
```

The tag `image:` designates a path of a docker image. When setting up a new course, you may need to change the path specified here

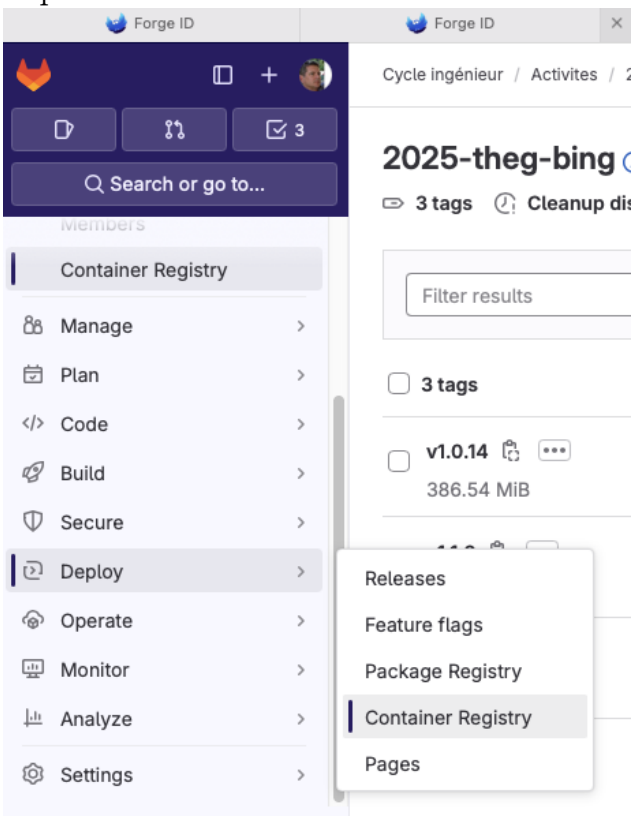
The correct `image:` path comes from the `gitlab.cri.epita` Container Registry and is different for each course and tenant. *E.g.*, in gitlab, navigate the menus: `Deploy` / `Container Registry`.

The tag `memory_mb:` controls how much memory is allocated to the student for the process computing this assignment. You must specify this number in two places as there are two `memory_mb:` keywords in the file.

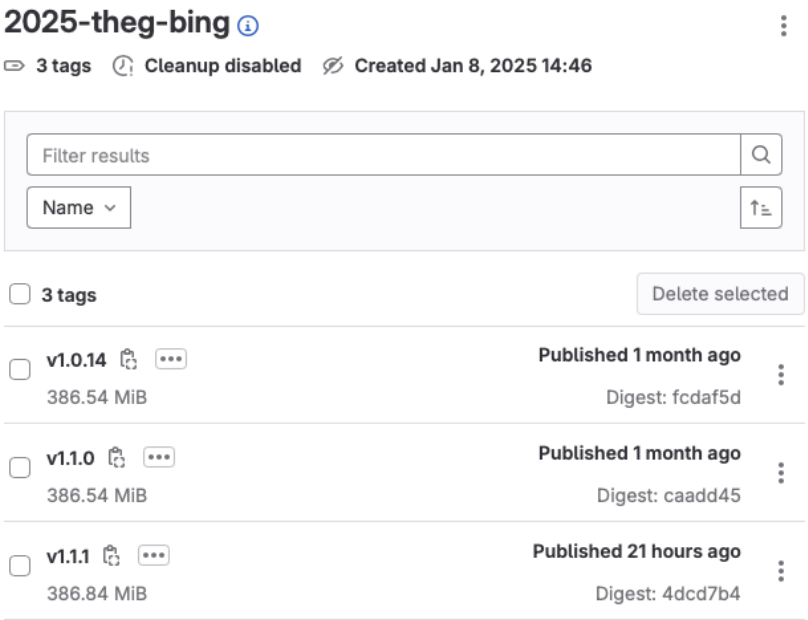
The tag `timeout` controls how much time is allocated to the student for the process computing this assignment. You must specify this number in two places as there are two `timeout:` keywords in the file.

If either the memory or time is exceeded (as per `memory_mb:` and `timeout:`), there may be no indication of that to the student. The student may see that it just never finishes, or that it fails with no error message and no

explanation.

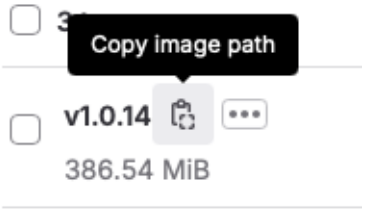


Clicking on `Container Registry` reveals a page like the following.



There is a `[copy]` button beside each version number. Click there to copy

the image path.



The value used for the `image:` path can take many different forms. Here are a few.

```
registry.cri.epita.fr/ing/activites/2025-clojure-elective-promo-2027
registry.cri.epita.fr/ing/activites/2025-scala-elective-promo-2027
registry.cri.epita.fr/ing/activites/2025-theg-bing
registry.cri.epita.fr/laboratoires/lab-si/tenants/epita/apprentissage/spring-2024-apfon
registry.cri.epita.fr/laboratoires/lab-si/tenants/epita/ing/activities/2024-folo
registry.cri.epita.fr/pi/activities/2025-linear-algebra
```

3.4.14 Directory `docker/`

The `docker/` directory contains files for constructing the docker image in which the student code and the auto-grader will be run.

3.4.15 Genericizing the Docker Harness

Several files in `docker/` are byte-identical across every course: `docker/xml-error.csh` (Section 3.4.19), `docker/fix-xml.py` (Section 3.4.20), `docker/docker-entrypoint.sh` (Section 3.4.17), and the generic-logic halves of `docker/launch_testsuite.csh` (Section 3.4.18) and `copy-tests.csh` (Section 3.4.9). Rather than hand-maintaining copies of these in every course's moulinette repository, the canonical copies live in `forge-flow/scripts/docker/`, and each course's `sync-docker-harness.csh` (a small, itself byte-identical script at the moulinette repository's root) refreshes its local `docker/` from there as the first step of the `docker-image` Makefile target (Section 3.4.1), ahead of `copy-tests.csh`.

This sync must happen, and be committed and pushed, *before* the git tag that triggers the CI docker build (i.e. before `new-version.csh` is invoked) – recall from Section 3.4.16 that the running container is offline, so every file the harness needs must already be physically present in `docker/` at build time, not fetched at grading time.

Two files can never be fully generic, since there is no way to make "how do I invoke pytest" vs. "how do I invoke `lein test`" vs. "how do I invoke `sbt testOnly`" itself a configuration variable:

- `docker/launch_testsuite.csh` keeps only its CONFIGURATION block (the `submission_specs` and `test_suffix` declaration) and sources the synced `docker/_lib_launch_testsuite.csh` for everything else. That generic file, in turn, sources `docker/run_tests.csh` – a short, course-specific, *not* synced file containing only the one or two lines that actually invoke the test runner – at the point where tests need to run.
- `copy-tests.csh` (at the moulinette repository's root, not in `docker/`) keeps only its CONFIGURATION block (the Professor Repository URL, the registry course/year, and the `sync_specs` declaration of what to copy where) and sources the synced `docker/_lib_copy_tests.csh` for everything else.

When bootstrapping a new course, copy `sync-docker-harness.csh` byte-identical from `forge-flow/scripts/sync-docker-harness.csh`, then run it once to populate `docker/`; write the new course's own CONFIGURATION blocks in `docker/launch_testsuite.csh` and `copy-tests.csh`, and its own `docker/run_tests.csh`. A generic-logic bugfix made later to a canonical file in `forge-flow/scripts/docker/` then reaches every course automatically on that course's next `make docker-image`.

3.4.16 File `docker/Dockerfile`

There are two phases of populating the Docker image. The first phase is done while the image is being constructed; the second phase is done every time a student assignment is graded. The file `docker/Dockerfile` contains instructions which are performed once, while a version of the image is being constructed. During this phase the code has access to the Internet and can download files. While student code is running, there is no Internet access. Only files which have been previously downloaded are reachable.

The `docker/Dockerfile` establishes `docker/docker-entrypoint.sh` as the `ENTRYPOINT`, *i.e.*, the file which will be executed each time the auto-grader sequence begins, once per student submitted assignment.

3.4.17 File `docker/docker-entrypoint.sh`

Byte-identical across every course, synced from a canonical copy – see Section 3.4.15.

The `docker/docker-entrypoint.sh` is our hook into the auto-grader. Its job is to extract the student submission files from a tarball using `tar xvf`. The student files (all the files associated with the `git tag` which the student committed during the `git push`, `git push -tags` phase of submission to Forge/Intranet.

The student's files are extracted into a directory named `/student/`. Thereafter, the script `docker/launch_testsuite.csh` is launched (Section 3.4.18) with the argument coming from `command:` tag in the corresponding `workflows/*.yml` file.

In some cases after `launch_testsuite.csh` is finished, there are several `JUnit`-style XML files which need to be combined into a single `/output.xml` file.

3.4.18 File `docker/launch_testsuite.csh`

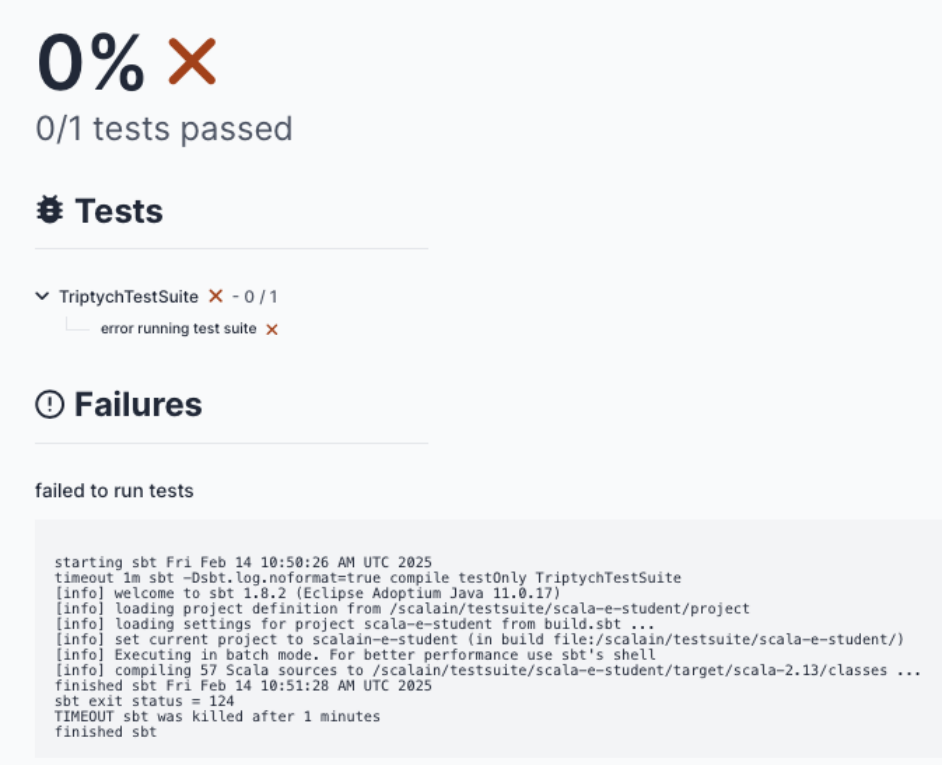
Only the CONFIGURATION block and the language-specific test invocation are course-specific; the generic logic described below is synced from a canonical copy – see Section 3.4.15.

The `docker/launch_testsuite.csh` script is launched by `docker/docker-entrypoint.sh` (Section 3.4.17) after the student's submitted files have been extracted into a directory named `/student/`. `docker/launch_testsuite.csh` is also given an argument which indicates which test suite needs to be run. This test suite indicates a test class name or file name depending on the programming language being tested. The indicator of the test suite comes from the `command:` target from the corresponding `.yml` file in the `workflows/` directory.

The job of `docker/launch_testsuite.csh` is to take the student files from `/student/` and move them into the place where the programming language specific testing environment can access them. For example to run the Scala tester, the files need to be in a very specific and rigid directory structure so `sbt test` can find them. `docker/launch_testsuite.csh` arranges the files into such a directory structure, and then launches the language-specific testing engine. Various such engines are discussed in Section 3.3.

In setting up to run the testing engine, `docker/launch_testsuite.csh` has access to all the files within `docker/suite`, Section 3.4.21.

One of the challenging tasks of constructing the `docker/launch_testsuite.csh` is error handling. If the student's code encounters an error when loading or encounters an exception other than an explicit test failure, we still want to give feedback to the student.



To report errors in a way which is compatible with the `JUnit` XML format, `docker/launch_testsuite.csh` should call the `./xml-error.csh` script, Section 3.4.19.

3.4.19 File `docker/xml-error.csh`

Byte-identical across every course, synced from a canonical copy – see Section 3.4.15.

The `docker/launch_testsuite.csh` script may call the `./xml-error.csh` script with two arguments to report errors which are correctly incorporated into the `output.xml` file. The output of `xml-error.csh` should be appended to `/output.xml` as in the following example.

Listing 3.14 (*Example of* `xml-error.csh`)

```
1 foreach dir (student)
2   if (! -d $dir ) then
3     echo "missing directory $root/$dir" | STDERR
4     ./xml-error.csh "internal-error" \
5                     "missing directory $root/$dir" \
6         >> /output.xml
7     exit 0
8   endif
9 end
```

The two command line arguments given to `xml-error.csh` indicate a short *error name* and a possibly long `error description`.

3.4.20 File `docker/fix-xml.py`

Byte-identical across every course, synced from a canonical copy – see Section 3.4.15.

In each student programming language used in this flow, the Forge framework is expecting to find a file named `/output.xml` at the top level after the student code has been tested. This XML file is generally defined by some `JUnit` standard, but all interfaces we have used fail to create the `/output.xml` file in the exact format which Forge expects.

We have created a program `fix-xml.py` which can read many XML formats and convert to the format which Forge expects.

The `JUnit`-style XML output of whichever testing framework is being used should be passed through the `fix-xml.py` program to produce the final `/output.xml` file. The program is idempotent in the sense that if the file is already in the correct format, then there is no harm in passing it through the program yet again.

This script finds many common discrepancies and repairs them in a way that the student-facing web page can display meaningful error messages.

3.4.21 Directory `docker/suite`

The directory, `docker/suite`, can be populated as part of the `make docker-image` target in the top level `Makefile` of the moulinette repository, Sections 3.4.1 and 3.4.9.

This directory is normally used as a place to run the programming language-specific testing framework as discussed in Section 3.3. Such a framework nor-

mally expects a certain directory structure to exist with configuration files, source files (files to be tested), and test case files. Most of these files already exist in the professor repository because it is already used to test the code and the test cases. The `copy-tests.csh` script (Section 3.4.9) copies this directory structure from the professor repository to the `docker/suite` directory.

The `docker/suite` directory may also need some files which require Internet access to create such as libraries or other compiled code. If so, these files should be generated in the file `docker/Dockerfile` (Section 3.4.16).

For example, certain files are needed for Clojure which are created by the lines in `docker/Dockerfile`.

Listing 3.15 (*Setup Clojure Tester in `Dockerfile`*)

```
WORKDIR suite
RUN lein deps
RUN lein test common.util-test
RUN rm -f TEST-common.util-test.xml
```

Similar files are created for Scala by the lines:

Listing 3.16 (*Setup Scala Tester in `Dockerfile`*)

```
WORKDIR /scalain/testsuite/scala-e-student
RUN sbt update
```

3.5 Computing Grades

Grade computation is performed by per-course scripts that live *outside* the moulinette repository, in `maas-courses/courses/` (Section 2.2). The moulinette's `compute_grades.clj` (Section 3.4.7) is deprecated.

Many colleagues enforce a hard deadline: late submissions receive a zero. That policy is simple to implement — `forge submissions success-report` gives the best raw score per student and requires no further processing. The system described here takes a different approach: a student who submits a few days late should not be penalized as heavily as one who submits at the last possible moment before the extended cut-off. This graduated late-penalty policy is the reason the grade computation is more involved: it requires the *complete* submission history (not just the best raw score) so that the best *penalized* score can be selected across all attempts, on-time and late alike.

The workflow has two steps:

1. Download the complete submission history for the course using `forge submissions tags-report`.
2. Run a per-course Clojure script that applies course-specific grading rules and writes a `grades.csv` file.

3.5.1 Forge CLI: Downloading Submissions

Authentication is required before any CLI call. See Section 2.3 for `forge auth token`.

The primary CLI call for grade computation is `forge submissions tags-report`, which downloads the *complete submission history* for the entire course in a single call.

Listing 3.17 (*CLI `forge submissions tags-report`*)

```
forge submissions tags-report \
    epita-pi-cs-bachelor/2026-linear-algebra \
    output.csv
```

This produces a CSV with one row per submission attempt (not per student), with the following columns.

```
id,status,error,ref,groupSlug,assignmentUri,
receivedAt,author,job_id,job_successPercent,job_status
```

The important columns are:

- `status` – one of four values:
  - `SUCCEEDED` – the grader ran to completion; `job_successPercent` is valid.
  - `ERROR` – student code crashed, failed to compile, or the grader itself failed; `job_successPercent` is meaningless.
  - `PROCESSING` – the grader was still running when the report was downloaded.
  - `IDLE` – the submission is queued but has not started.
- `ref` – the tag submitted by the student, *e.g.*, `Hello-2025-02-16T14-05-36`

- `assignmentUri` – identifies the assignment, *e.g.*, `epita-pi-cs-bachelor/2026-linear-algebra/root/Hello`
- `receivedAt` – timestamp the tag was received, *e.g.*, `2025-02-16T11:33:36.61648Z`
- `author` – student login
- `job_successPercent` – integer percentage score for this submission attempt (only valid when `status == SUCCEEDED`)

Before computing grades, filter the CSV to keep only rows where `status == SUCCEEDED`. Rows with any other status must be discarded: they either carry no meaningful score (`ERROR`) or the score is not yet final (`PROCESSING`, `IDLE`). Counting an `ERROR` row as a zero is incorrect — the student may have a valid `SUCCEEDED` submission for the same assignment submitted earlier or later.

Unlike `forge submissions success-report` which returns only the best grade per student per assignment, `tags-report` returns all attempts. This is necessary to correctly apply late penalties, since the best *penalized* score may differ from the best raw score.

`forge submissions success-report` is useful for quick one-off lookups of a single assignment.

Listing 3.18 (*CLI* *forge submissions success-report*)

```
forge submissions success-report \
  epita-pi-cs-bachelor/2026-linear-algebra \
  epita-pi-cs-bachelor/2026-linear-algebra/root/Hello \
  Hello.csv
```

3.5.2 Per-course Grade Computation Script

Each course has a `compute_grades.clj` that loads a shared library (`compute_grades_incl.clj`) and calls `gen-grades-csv` with course-specific parameters.

Listing 3.19 (*Example per-course* `compute_grades.clj`)

```
(load-file "../../compute_grades_incl.clj")

(gen-grades-csv
  {:tenant          "epita-pi-cs-bachelor"
   :course          "2026-linear-algebra"
   :forge-url       "gitgitlab.cri.epita.fr:pi/activities/2026-linear-algebra.git"
["Eigenvalues" "Final""Late-Eigenvalues" "Late-Final"]:late-prefix
"Late-":ignore-prefix "Prof-":default-penalty 0.6:allow-days-late
30.0)
```

The shared library automatically clones the moulinette repository (via `:forge-url`) to read `activity.json` for assignment dates and the student list. It then calls `forge submissions tags-report` and computes each student's best penalized score across all submission attempts.

The key parameters are:

- `:tenant`, `:course` — identify the course to the `forge` CLI.
- `:forge-url` — the moulinette repository URL, cloned to obtain `activity.json`.
- `:skip` — assignments to exclude from the grade average, *e.g.* finals or unfinished assignments.
- `:late-prefix` — prefix used for late submission tags, typically `"Late-"`.
- `:ignore-prefix` — tag prefix to exclude entirely, typically `"Prof-"`.
- `:default-penalty` — the penalty coefficient applied to late submissions, *e.g.* `0.6` means a 40% penalty.
- `:allow-days-late` — enables a sliding penalty window (see below).
- `:students-wo-penalty` — set of student logins exempt from late penalties.
- `:assignments-wo-penalty` — set of assignments exempt from late penalties.

3.5.3 Sliding Late Penalty

When `:allow-days-late` is greater than zero, the penalty scales linearly with lateness rather than applying the full `:default-penalty` immediately. For example, with `:default-penalty 0.6` and `:allow-days-late 30`:

- A submission on time receives a coefficient of `1.0` (no penalty).
- A submission 15 days late receives a coefficient of `0.8` (half the penalty).
- A submission 30 or more days late receives the full `0.6` coefficient.

3.5.4 Output

The script writes `grades.csv` with the following columns per student: `login`, `epita email`, `per 20` (final grade out of 20), `per 20 to-date` (grade based only on assignments whose opening date has passed), `total`, `percentage`, and for each assignment: the penalized score, days late, penalty coefficient, submission timestamp, and assignment closing date.

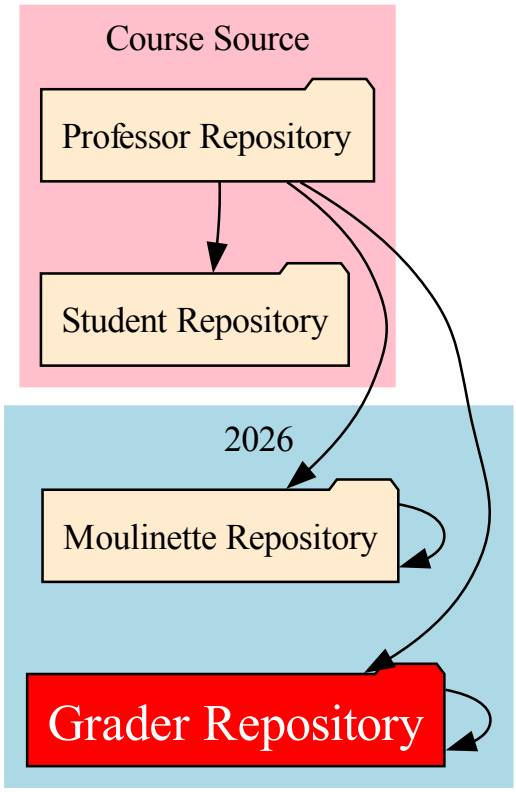


Figure 4: Grader Repository

4 Grader Repository

This repository mimics the student interface. Tagging and pushing this repository triggers the submissions to be autograded and their results to be displayed on the Forge web page.

The grader repository can be created by using `git clone`. It already exists assuming the moulinette repository (Section 3) has been created and pushed successfully. The URL to clone has been determined by the Forge/Intranet service setup. You may find the URL by interactively examining an assignment

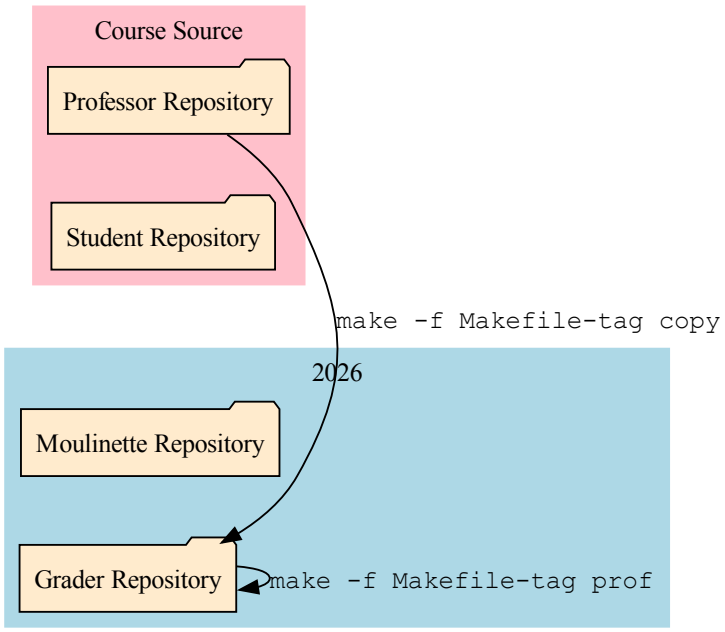


Figure 5: Grader Repository `Makefile-tag` Targets: `copy`, `prof`

submission page, search for the header `Repository`.

Forge/Intranet expects commits and tags to be found on the `master` branch of this repository. Tags on the `main` branch are a source of difficult-to-find errors in the flow. The moulinette repository (Section 3) is expected to have commits on the `main` branch. This `master`/`main` is a potential source of confusion.

- File `Makefile-tag` This Makefile is named `Makefile-tag`, because sometimes there may be a `Makefile` for the student to use. In order to avoid name conflicts, we have named this *administrative* makefile `Makefile-tag`.

Figure 5 illustrates the `Makefile-tag` targets in this repository. The grader repository has a top level makefile named `Makefile-tag` with these make targets.

1. `make -f Makefile-tag prof`
2. `make -f Makefile-tag copy`

- File `copy-files.csh`
- File `tag-prof.csh`
- File `check_activities.clj`

4.1 Target `make -f Makefile-tag copy`

The target `make -f Makefile-tag copy` copies the correct solutions of the homework assignments from the professor repository to the correct place in the grader repository. In some courses the homework solutions files all go at the top level of this repository, and in some courses they go into the directory structure. The `make -f Makefile-tag copy` target understands this and copies into the correct place. After the files are copied into place, the changes are (git) committed but not (git) pushed.

This make target is usually a call to the `copy-files.csh` shell script.

4.2 Target `make -f Makefile-tag prof`

This target is responsible for running the Forge tag/push flow, which includes `git tag ...`, `git push`, and `git push -tags`. The purpose is to test and verify that the proposed solutions in the professor repository actually satisfy the tests. *E.g.*, this test should detect whether the tests are wrong, or whether the proposed solutions are wrong, or whether the declarations in `activity.yml` are wrong.

The `make -f Makefile-tag prof` is typically implemented by running a script `tag-prof.csh`.

A challenge with implementing `make -f Makefile-tag prof` is that Forge does not allow even the professor to push a tag whose assignment is not in an open state. *I.e.*, you cannot push a tag for an assignment if that assignment date has not arrived or the assignment closing date has passed. For example, suppose the `activity.yml` file in the moulinette repository has the following section for an assignment.

Listing 4.1 (*Tag Declaration in* `activity.yml`)

```
    submissions:
      - slug: Hello
        autoPublish: true
        handoffType: GIT_TAG
        tagPrefix: Hello-
        limit: 60
        limitType: HOUR
        maxInFlight: 10
        pick: BEST
        dates: # Hello
          publishingAt: '2025-01-01T08:00:00+01:00'
          openingAt:    '2025-01-01T08:00:00+01:00'
          closingAt:    '2025-02-15T22:59:00+01:00'
```

Then normally a student must tag with a tag such as `Hello-2025-04-02-18-32`. However, this tag will be rejected if the current date is 2 April 2025, because the assignment closes on 15 February 2025 at 22h59.

As also explained in Section 3.4.3, if the dates for assignments in a course span the entire course, then `make -f Makefile-tag prof` should simply create a tag such as `Hello-2025-04-02-18-32`. However, if the dates are restricted, then there should be non-expiring tags with a `Prof-` prefix such as `Prof-Hello-2025-04-02-18-32`, and `make -f Makefile-tag prof` should create those tags instead.

`make -f Makefile-tag prof` should create one tag per assignment and push them all.

Before creating any new tags, all previous tags are deleted with `git tag -d ‘git tag‘`. This is necessary because there are git hooks in place which are very slow, and will refuse all tags if one single tag is incorrect. This includes refusal if there is an irrelevant tag. *I.e.*, using Forge/Intranet means you are not allowed to use any other tags in your repository.

After the tags have been created, the `make -f Makefile-tag prof` target runs the following:

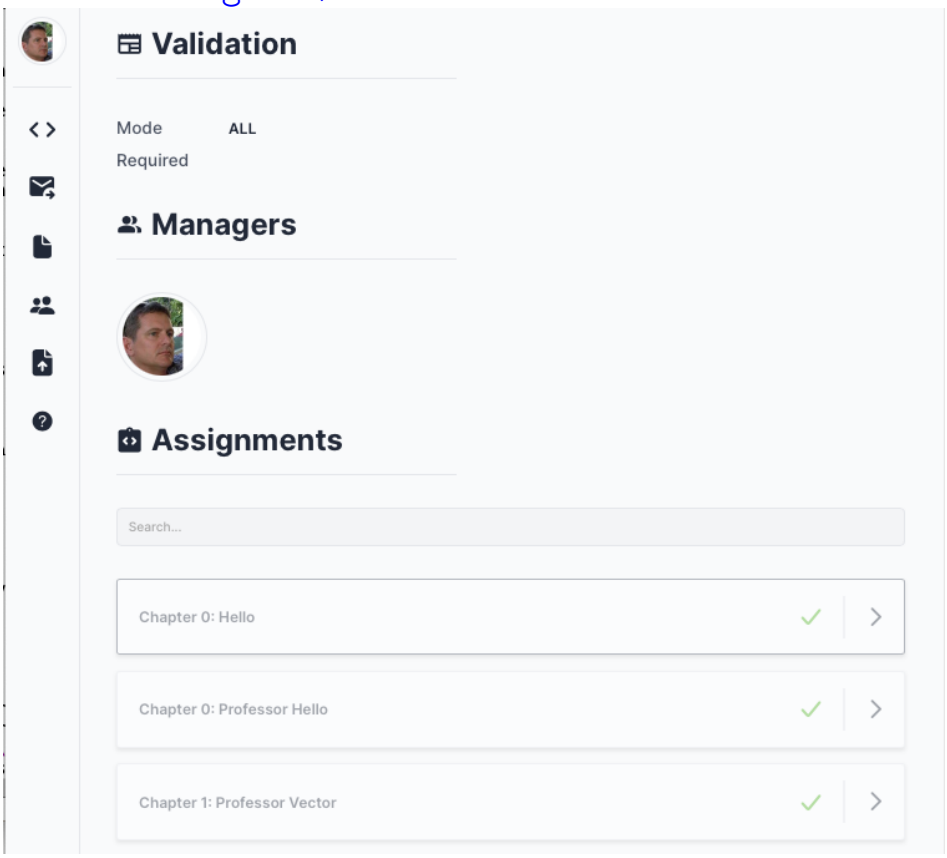
Listing 4.2

```
1   git push
2   git push --tags
```

After the tags have been pushed, the Forge engine starts evaluating the submissions in secure Docker images and eventually the results will be displayed

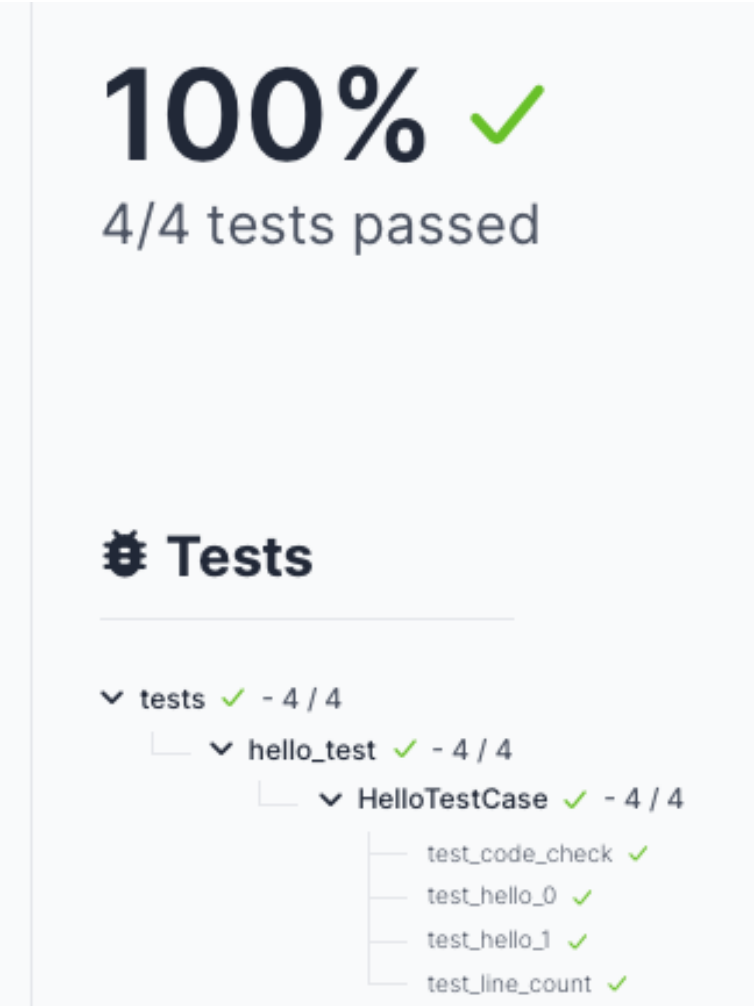
on the web page corresponding to the course.

E.g., https://intra.forge.epita.fr/epita-pi-cs-bachelor/2025-linear-algebra/root



Suppose some test fails; *i.e.*, suppose the newly tagged code fails a test, or suppose a test has been changed and the test itself is wrong. Then the green check marks may still appear on this web page because the page may be showing the `BEST` result, rather than the most recent `LAST` result. Therefore, you must verify that every test has passed. This would ordinarily be done by clicking through the menu web widgets to verify that all the most recent still

have a score of 100 percent.



To avoid this tedious, error-prone test, the `make -f Makefile-tag prof` target also runs a script which checks the most recent scores in batch. This is typically accomplished when the script `tag-prof.csh` runs the script `check_activities.clj`.

Typically output of this process looks something like the following. You see that if everything is correct, the status goes from `IDLE` to `PROCESSING` to `SUCCEEDED`. However, if there is an error the status will go to `ERROR` in which case the process will abort, and you need to debug why that particular tagged submission failed.

Listing 4.3 (*Output of* `check_activities.clj` *Script*)

```
[:status IDLE :tag Matrix-2025.02.12.13]
[:status PROCESSING :tag Hello-2025.02.12.13]
[:status IDLE :tag Field-2025.02.12.13]
[:status PROCESSING :tag Angle-2025.02.12.13]
[:waiting ...]

[:status SUCCEEDED :grade 100 :tag Angle-2025.02.12.13]
[:status PROCESSING :tag Vector-2025.02.12.13]
[:status PROCESSING :tag Matrix-2025.02.12.13]
[:status PROCESSING :tag Hello-2025.02.12.13]
[:status PROCESSING :tag Group-2025.02.12.13]
[:status PROCESSING :tag Field-2025.02.12.13]
[:waiting ...]

[:status SUCCEEDED :grade 100 :tag Hello-2025.02.12.13]
[:status SUCCEEDED :grade 100 :tag Group-2025.02.12.13]
[:status SUCCEEDED :grade 100 :tag Field-2025.02.12.13]
[:status PROCESSING :tag Vector-2025.02.12.13]
[:status PROCESSING :tag Matrix-2025.02.12.13]
[:waiting ...]

[:status SUCCEEDED :grade 100 :tag Vector-2025.02.12.13]
[:status SUCCEEDED :grade 100 :tag Matrix-2025.02.12.13]
```

Each process that goes into status `SUCCEEDED` will be marked with a grade, which is the percentage of tests which passed for that assignment. If everything is correct, this percentage will be `100`. However, if some percentage is less than `100`, the process will abort, and you will need to debug the proposed solution or the test case. The following is such an example where the grade was 22 percent rather than the expected 100.

Listing 4.4 (*Example Error Output of* `check_activities.clj` *Script*)

```
[:status SUCCEEDED :tag Primes-2025-12-02-16-45-36
                   :grade 22]
Execution error (AssertionError) at
  check-activities/eval261 (check_activities.clj:106).
Assert failed: Grade 22 < 100 for Primes-2025-12-02-16
(= "100" (get hash "job_successPercent"))
```

If the job fails spectacularly, you may need to debug using the operator. https://operator.forge.epita.fr/ui/

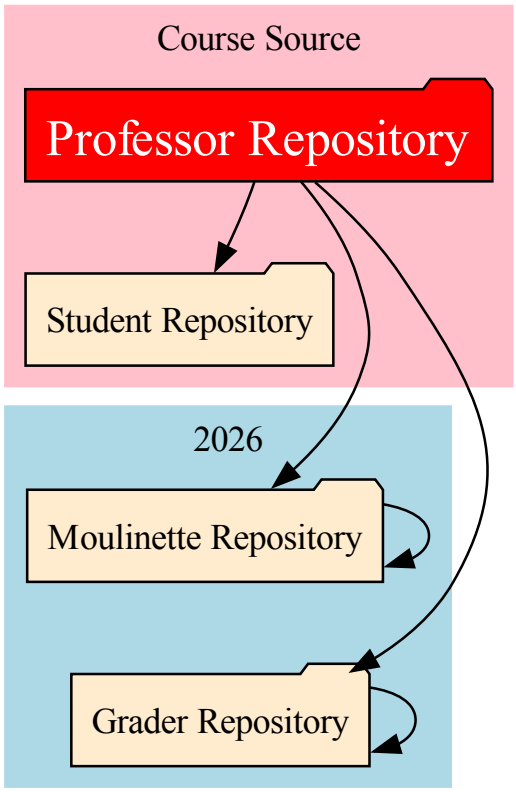


Figure 6: Professor Repository

5 Professor Repository

This repository is the development area for the professor. The same professor repository may be used in association with several different courses. *E.g.*, the same course might occur once per year, in which case a different moulinette repository is created each year, but the same professor repository is reused year after year. Additionally, the same professor repository might be associated with several different courses. *E.g.*, the professor repository `scalain_e`, whose URL is `git@gitlab.cri.epita.fr:jim.newton/scalain_e.git`, is used for several courses: (SCALAIN) Scala Elective in the ING-1 curriculum, (APFON) Applications in Functional Programming in the EPITA-Apprentissage curriculum.

This repository contains confidential information not available to students such as solutions to assignments as well as development history.

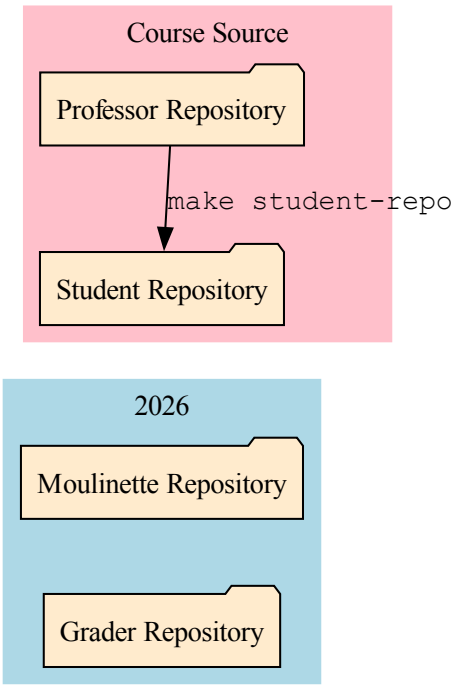


Figure 7: Professor Repository `Makefile` Target: `student-repo`

5.1 File `Makefile`

Figure 7 illustrates the `Makefile` target `student-repo`.

This `Makefile` has a target `student-repo` which filters certain files from the professor repository into files in the student repository (Section 6).

5.2 Filtering Student Assignments

This topic addresses the problem of how to keep the solutions to student exercises in sync with the student exercises themselves. From time to time the professor will need to modify test cases, fixing errors in test cases, or adding additional test cases to cover untested cases. It may also be necessary to revise the solutions to make sure they are using good techniques and actually testing what they need to test; *i.e.* prevent accidentally passing the tests.

For this reason, we recommend implementing the solutions in the professor repository, so that the student does not have access to them, and auto-generate the student templates which include the function names, argument lists, type annotations and assertions. These auto-generated templates will omit the parts of the code which the student needs to insert to make the tests pass. How to do this templating and filtering differs by programming language as programming language syntax varies from language to language.

The sample solutions must run as they are and pass all the tests, and we must be able to extract the student templates fully automatically from them.

We have prepared three filtering mechanisms for the three languages we use. If we add new languages, we'll have to come up with a way to implement the templating in the new language.

- Section 5.2.1: `filter-python-challenges.py`
- Section 5.2.2: `filter-clojure-challenges.py`
- Section 5.2.3: `filter-scala-challenges.py`

5.2.1 File `filter-python-challenges.py`

It is important that the professor be able to test the *control* code to make sure the tests pass with the *ideal* solutions. In our case we derive the student templates by filtering these ideal solutions and removing certain pieces of code. The script `filter-python-challenges.py` reads the Python files in the directory named on its command line,

and produces filtered files in the other directory named on its command line. For example, it will convert from the following function containing `# CHALLENGE: student must complete the implementation.` and generate text of the function hereafter, having code replaced with `raise NotImplementedError()`.

Listing 5.1 (*Example Pre-filtered Python Code*)

```
def prime_factors(n: int) -> List[int]:
  def loop(start, n):
      if n == 1:
          return []
      # find a prime factor of n
      p = smallest_factor(n, start, ceil(sqrt(n)) + 1)
      if p == n:
          # then n is prime
          # CHALLENGE: student must complete the implementation.
          return [n]
      else:
          # then n is composite and its smallest prime factor is p
          # CHALLENGE: student must complete the implementation.
          return [p] + loop(p, n // p)

  return loop(2, n)
```

The following is the code after filtering; notice the insertion of `raise NotImplementedError()`. When `filter-python-challenges.py` sees the line `# CHALLENGE: student must complete the implementation.`, it skips all the following lines until it finds an indentation more to the left from the current line. It inserts a comment about how many lines were skipped. This comment serves as an indication to the student of approximately how many lines of coding are required to complete the exercise.

Listing 5.2 (*Example Post-filtered Python Code*)

```python
def prime_factors(n: int) -> List[int]:
    def loop(start, n):
        if n == 1:
            return []
        # find a prime factor of n
        p = smallest_factor(n, start, ceil(sqrt(n)) + 1)
        if p == n:
            # then n is prime
            # CHALLENGE: student must complete the implementation.
            # HINT: goal = 1 line
            raise NotImplementedError()

        else:
            # then n is composite and its smallest prime factor is p
            # CHALLENGE: student must complete the implementation.
            # HINT: goal = 1 line
            raise NotImplementedError()

    return loop(2, n)
```

5.2.2 File `filter-clojure-challenges.py`

Similar to `filter-python-challenges.py` (Section 5.2.1) except it works for Clojure code. There is an important restriction with how `filter-clojure-challenges.py` works. When the text `#_challenge` is found, `filter-clojure-challenges.py` simply omits everything from the leading `#` to end of line; it does not count parentheses. Therefore, care must be taken so that parentheses will still match after the rest of the line is omitted. For example if the line looks like this, `filter-clojure-challenges.py` will create invalid code, as it will delete too many closing parentheses.

Listing 5.3 (*Clojure Challenge Declaration, losing parentheses*)

```
        #_challenge (- remaining-size piece-size))))
```

To avoid this problem, the professor must write the code with this awkward limitation in mind. The code should look like the following instead, to assure that only matching parentheses are omitted.

Listing 5.4 (*Clojure Challenge Declaration, retaining parentheses*)

```
        #_challenge (- remaining-size piece-size)
        )))
```

The inclusion of `#_challenge` in the professor's code does not affect how the code behaves. The Clojure reader interprets the characters `#_` as a notation to simply ignore the following expression, whether that expression is a symbol, number, string, or parenthesized expression such as `#_"hello"`, `#_1234.566`, or `#_( this list [is ignored])`.

The following example shows the before and after with respect to `#_challenge`.

Listing 5.5 (*Example Pre-filtered Clojure Code*)

```
(defn piece-sizes-rec []
  (loop [n 1
         remaining-size 1.0
         acc ()]
    (let [piece-size #_challenge (* (/ n 100.0) remaining-size)
          ]
      (if (= n 100)
        #_challenge (reverse (sort-by second
                      (conj acc [n remaining-size])))
        (recur #_challenge (inc n)
               #_challenge (- remaining-size piece-size)
               #_challenge (conj acc [n piece-size])
               )))))
```

The following is the result of the above code after being filtered by `filter-clojure-challenges.py`.

Listing 5.6 (*Example Post-filtered Clojure Code*)

```
(defn piece-sizes-rec []
  (loop [n 1
         remaining-size 1.0
         acc ()]
    (let [piece-size (throw (ex-info
                  "Missing single expression, not yet implemented" {}))
          ]
      (if (= n 100)
        (throw (ex-info
                  "Missing single expression, not yet implemented" {}))
        (recur (throw (ex-info
                  "Missing single expression, not yet implemented" {}))
                  (throw (ex-info
                  "Missing single expression, not yet implemented" {}))
                  (throw (ex-info
                  "Missing single expression, not yet implemented" {}))
                  )))))
```

The marked expression must also be present on the *same* line as `#_challenge` itself. If the marker is the last token on a line, with the expression it marks beginning on the following line instead, `filter-clojure-challenges.py` used to do nothing at all to that line: no throw stub was substituted, and the professor's real solution was emitted into the student-facing file completely unredacted. The script now detects this and refuses to proceed, identifying the exact file and line:

Listing 5.7 (`#_challenge` *with no expression on the same line*)

```
ValueError: factors.clj: unsupported '#_challenge' usage
   (marker not followed by an expression on the same line):
   '            #_challenge\n'
```

The `filter-clojure-challenges.py` program can also filter a second kind of syntax. Lines containing `;; CHALLENGE:` are filtered away along with all the following lines indented to the right. Once a line is encountered with the same indentation or indentation to the left, the suppression is terminated.

Listing 5.8 (*Example Pre-filtered Clojure Code*)

```
(defn derivative+ [f a dx epsilon]
  (let [fa (f a)
        slope (fn [dx] (/ (- (f (+ a dx)) fa)
                          dx))]
    ;; CHALLENGE: student must complete the implementation.
    (loop [s (slope dx)
           dx (/ dx 2.0)]
      (let [s' (slope dx)]
        (if (< (abs (- s s')) epsilon)
          s'
          (recur s'
                 (/ dx 2.0)))))
    ))
```

Notice that all the lines after `;; CHALLENGE:` which are indented to the right are replaced with the call to `(throw ...)` and a comment indicating how many lines were suppressed. This comment gives the student an idea of approximately how many lines are needed to complete the exercise.

Listing 5.9 (*Example Post-filtered Clojure Code*)

```
(defn derivative+ [f a dx epsilon]
  (let [fa (f a)
        slope (fn [dx] (/ (- (f (+ a dx)) fa)
                          dx))]
    ;; CHALLENGE: student must complete the implementation.
    (throw (ex-info
       "Missing one or more expressions, not yet implemented" {}))
    ;; HINT 7 line(s)
    ))
```

Again, care must be taken with forms containing `;; CHALLENGE:` because `filter-clojure-challenges.py` does not count parentheses; it only looks at lines and indentation. An example like the following would result in too many closing parentheses to be omitted, because as it is written, the closing parenthesis of the `(let [a 100]` is at the end of the line `(+ a b)))`, thus too many closing parentheses would be omitted.

Listing 5.10 (*Pre-filtered Code with ERROR, Omitting too many Parentheses*)

```
(let [a 100]
  ;; CHALLENGE: student must complete the implementation.
  (if (> b a)
     (* a b)
     (+ a b)))
(recur (rest data))
```

It is important, in this case, to put the closing parenthesis of the let on its own line with correct indentation, as follows, either at the same indentation of `;; CHALLENGE:` or to the left thereof.

Listing 5.11 (*Pre-filtered Code, Fixing Parenthesis error*)

```
(let [a 100]
  ;; CHALLENGE: student must complete the implementation.
  (if (> b a)
     (* a b)
     (+ a b))
  )
(recur (rest data))
```

Similarly, `;; CHALLENGE:` must be the first non-whitespace content on its line. The skip-until-matching-delimiter logic uses the column the semicolon sits at as the marked block's reference indentation; if real code precedes the marker on the same line (e.g. `(2) (- ;; CHALLENGE: ...`), that column no longer matches the surrounding code's actual structural indentation, and the skip-termination logic misfires the same way as the parenthesis- counting hazard shown above. This too is now detected immediately, rather than left for the professor to catch by inspection:

Listing 5.12 (*`;; CHALLENGE:` not first on its line*)

```
ValueError: determinant.clj: unsupported ';; CHALLENGE:' usage (marke
is not the first non-whitespace content on its line):
'               (2) (- ;; CHALLENGE: student must complete the
implementation.\n'
```

Finally, after writing each output file, `filter-clojure-challenges.py` re-reads it with Clojure's own reader, without evaluating it, to confirm the file as a whole is still syntactically well-formed. This is a deliberate second, independent safety net: the two positional checks above only catch the two specific

marker misuses described earlier, not every way a substitution can unbalance parentheses – the parenthesis-over-deletion hazards shown earlier in this section have the marker positioned correctly, so only this reader check catches those. A naive parenthesis count in Python cannot reliably perform this check either: a `)` character appearing inside a Clojure string literal is not a real closing delimiter, and only an actual Clojure reader can tell the difference. On failure, the script reports the line and column the reader had reached when it gave up:

Listing 5.13 (*Reader validation catching an unbalanced substitution*)

```
ValueError: geodesic.clj: generated file (.../geodesic.clj) failed to
parse as valid Clojure at line 104, column 6:
java.lang.RuntimeException: Unmatched delimiter: ) -- likely
mismatched parens from a #_challenge/CHALLENGE substitution.
```

5.2.3 File `filter-scala-challenges.py`

Similar in concept to `filter-python-challenges.py` (Section 5.2.1) except it works for Scala code. The `filter-scala-challenges.py` program does two types of suppression of code.

1. If it finds `// CHALLENGE:` followed by one or more commented lines, then uncomments the lines and omits an equal number of lines following that. *I.e.*, the commented lines replace the lines of code. Here is an example:

   Listing 5.14 (*Pre-filtered Scala Code*)

   ```
   def makeAdjDirected[V](edges: Seq[(V,V)]):Map[V,Set[V]] = {
     // CHALLENGE:
     //val f1:((V,V))=>V = ???
     //val f2:((V,V))=>V = ???
     val f1:((V,V))=>V = pair => pair._1
     val f2:((V,V))=>V = pair => pair._2
     edges.groupBy(f1)
       .map{case (src, edges) =>
         val destinations = edges.map(f2)
         src -> destinations.toSet
       }
   }
   ```

   Notice that the lines of code which were commented above are no longer commented, and they replace the same number of lines from the original code.

Listing 5.15 (*Post-filtered Scala Code*)

```
def makeAdjDirected[V](edges: Seq[(V,V)]):Map[V,Set[V]] = {
  val f1:((V,V))=>V = ???
  val f2:((V,V))=>V = ???
  edges.groupBy(f1)
    .map{case (src, edges) =>
      val destinations = edges.map(f2)
      src -> destinations.toSet
    }
}
```

2. If it finds `// CHALLENGE/END:` then it omits that line followed by all following lines until it finds `// END`. A token `???` is inserted to replace the omitted lines. Here is an example:

Listing 5.16 (*Pre-filtered Scala Code*)

```
def geodesic_distance(loc1: Location, loc2: Location):
    Double = {
  // km distance between two (lat, lon) points
  val ((lat_1, lon_1), (lat_2, lon_2)) = (loc1, loc2)
  val phi_1 = lat_1.toRadians
  val phi_2 = lat_2.toRadians
  val lambda = (lon_1 - lon_2).toRadians

  (earth_radius *
    // CHALLENGE/END: student must complete the
       implementation.
    acos(sin(phi_1) * sin(phi_2)
          + (cos(phi_1) * cos(phi_2) * cos(lambda))
        )
    // END
    )
}
```

This section of code will be written as follows.

Listing 5.17 (*Post-filtered Scala Code*)

```
1  def geodesic_distance(loc1: Location, loc2: Location):
       Double = {
2    // km distance between two (lat, lon) points
3    val ((lat_1, lon_1), (lat_2, lon_2)) = (loc1, loc2)
4    val phi_1 = lat_1.toRadians
5    val phi_2 = lat_2.toRadians
6    val lambda = (lon_1 - lon_2).toRadians
7
8    (earth_radius *
9      ???
10     )
11 }
```

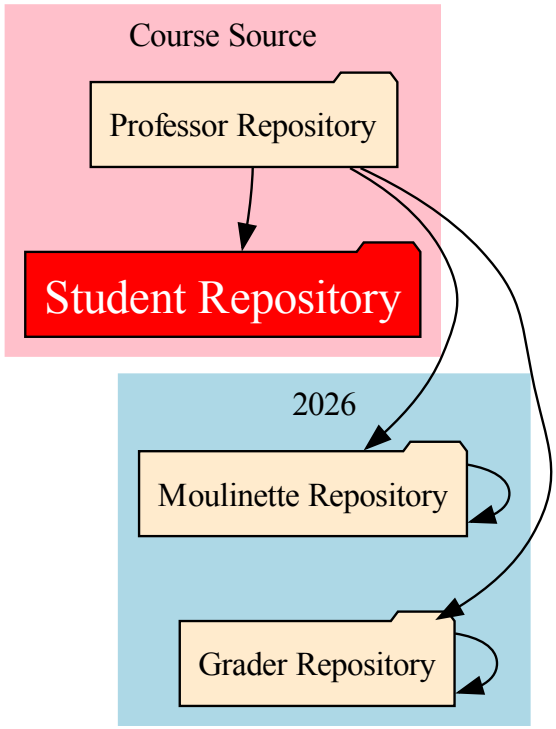


Figure 8: Student Repository

6 Student Repository

This repository is the student's view of the course. The purpose of the student repository is to provide the students with class notes, supplementary files, homework exercises *etc.* If the professor does any live coding in class, these files should get propagated to the student repository as well. These live coding examples should be visible to the student after the student performs `git pull`.

This repository is created mostly automatically by extracting information from the professor repository via the `make student-repo` target in the professor repository `Makefile`. In some cases the `make student-repo` target is implemented as a shell script in an underlying `bin` directory, and sometimes the code is directly in the `Makefile`.

It is important that the student repository have the templates for the homework exercises but not the answers, neither in the repository files nor in the revision history. The student templates are derived automatically by stripping out certain lines or certain expressions from the proposed solution files. There

are scripts for performing this filtering. The scripts are found in the professor repository and are explained in Section 5.2.

It is important (from the student's perspective) that this resource be a `git` repository, as from time to time during the course the professor needs to update some of the files, fix bugs, augment lecture notes and examples, add additional tests *etc.* The students are responsible for regularly running `git pull` to make sure their checked-out repository is up to date.

The student repository is readable to students but not writable to students. Students are responsible for cloning this repository to obtain assignments and lecture notes. This cloning may be done explicitly in the case of advanced students, or via the script `checkout-workarea.py` (Section 7.5) for more beginner students, depending on the workflow of the course.

Part of the automatic process of updating (auto-generating) the content of the student repository is creating the templates for the student assignments. Generally student programming assignments contain function declarations and partial implementations which trigger a *not-yet-implemented* exception. The responsibility of the student is to replace all such exceptions with correct code which fulfills the assignment and passes the tests. Usually, there is a comment at the point where the *not-yet-implemented* exception is thrown which gives the student a clue about approximately how many lines of code are necessary to complete the task.

There are several scripts in the bin directory which will be copied from the professor repository (the golden copy) to the student repository during the `make student-repo` target. These scripts are intended for the student to run to help manage populating and updating the workarea as well as submitting to the auto-grader.

The scripts are shown here but are explained in subsections of Sections 7 and 8.

- `checkout-workarea.py` (Section 7.5)
- `test-git.py` (Section 8.1)
- `test-ssh.py` (Section 8.2)
- `submit-to-grader.py` (Section 8.3)
- `import-to-workarea.py` (Section 8.4)

7 Student Initial Setup

The EPITA engineering curriculum provides student assistants to help other students navigate their usage of Forge/Intranet, but as of the writing of this document (14 Feb 2025), the Forge group provides no maintenance service. If students have problems with the system, either in using it or setting it up, there is no help available from Forge, it is 100% the professor's responsibility. This of course provides a huge challenge, and a great deal of time can be lost in courses trying to debug problems dealing with SSH keys, git installation on student Windows laptops, students using laptops set up in languages other than English or French.

Some students arrive in class having already used Forge/Intranet for projects. They have the necessary software already installed on their individual laptops such as `git` and `ssh`. These students already know how to use `git clone`, `git tag`, `git push`.

Some students arrive in class with no prior experience. This is particularly the case for international students in the Bachelor of Computer Science program. The following sections explain for these students how to set up `git` and how to register their `ssh` keys for using Forge/Intranet.

7.1 Public Keys

The students should have `git` and `ssh-keygen` already installed onto their laptops before coming to class. However, this is not always the case.

7.1.1 Installing `ssh` and `git`

Most systems come with these tools already installed. However, if you don't already have `git` and `openssh`, you need to install them on your laptop. You will have to install: `git` and `openssh` (at least the ssh client).

To install `git` on Windows, you can use the `git-for-windows` installer https://gitforwindows.org/.

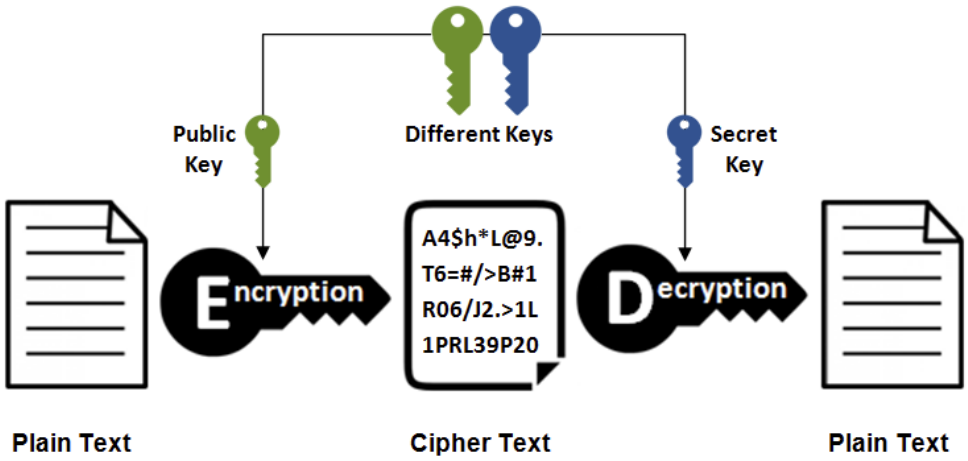
On Linux, `apt-get install git` suffices on Debian-based systems, and `pacman -S git` on Arch-based systems.

On MacOS, typing `git` will trigger the installation if it is not available.

You can find more information here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

The SSH client is already embedded in MacOS and Linux. To enable it on Windows, you can follow https://learn.microsoft.com/fr-fr/windows/

terminal/tutorials/ssh



7.1.2 What is `ssh-keygen`

According to https://www.techtarget.com, the `ssh-keygen` command is a component of most SSH implementations used to generate a public key pair for use when authenticating with a remote server. In the typical use case, users generate a new public key and then copy their public key to the server using SSH and their login credentials for the remote server.

By default, `ssh-keygen` creates an RSA key pair and stores the public key in a public key file named `.ssh/id_rsa.pub` and a private key file named `.ssh/id_rsa`.

7.2 Create a Public Key

To use the *intra* service you'll need to create an SSH public key and register that key in two places, once on https://cri.epita.fr (Section 7.3) and once on https://gitlab.cri.epita.fr (Section 7.4).

The procedure for creating an SSH public key differs across Windows, Linux, and MacOS.

7.2.1 From Windows

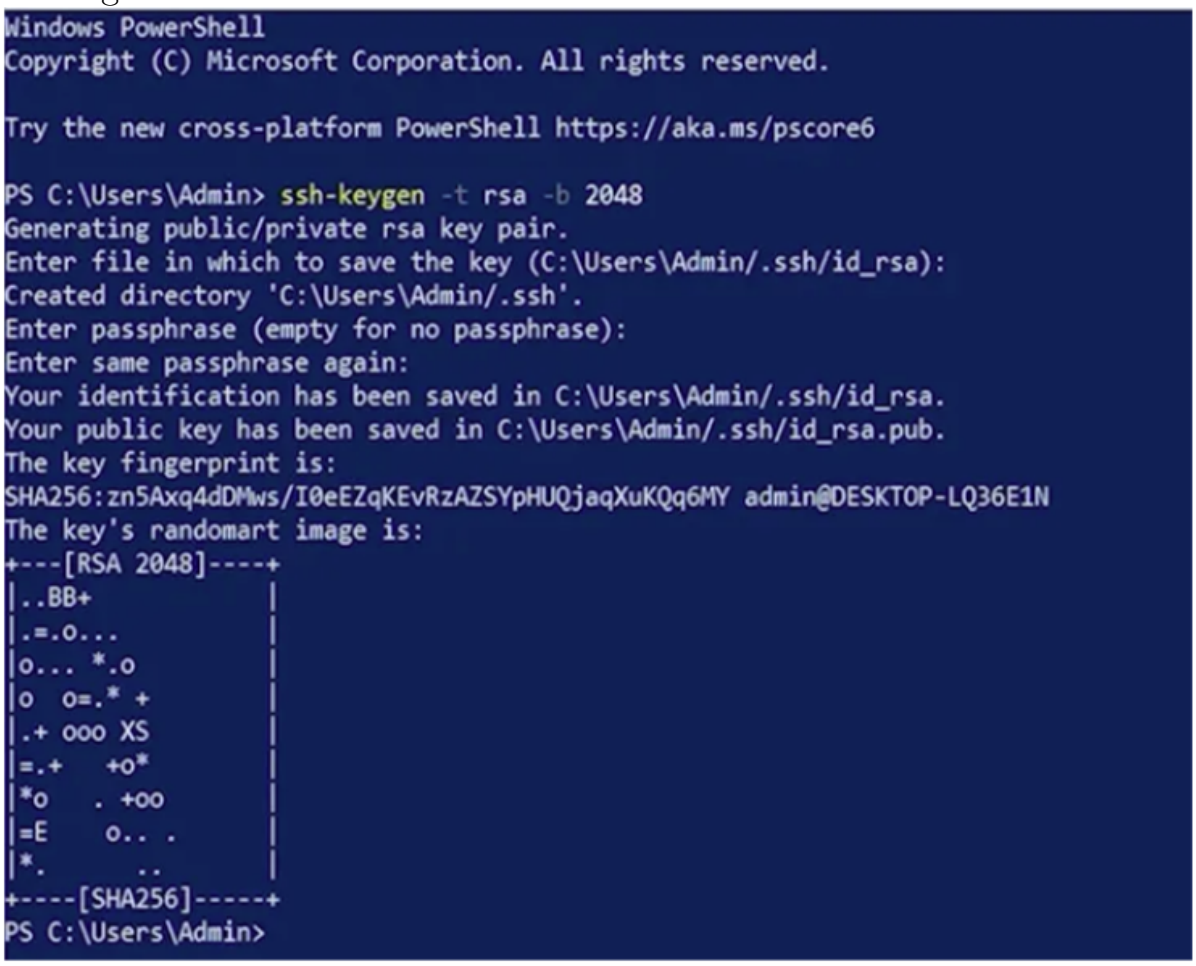
If you are using Windows, then you'll need to open the `powershell`.

From the Windows search bar, type `powershell` and click on the top result. A `powershell` window will open.

Next type `cd .ssh`. Use the command `ls` to discover the files in the `.ssh` directory. If you already have a file named `id_rsa.pub`, then you already have

a public key. Otherwise enter the command `ssh-keygen`; you will be prompted for several questions—press ENTER to accept the defaults for each prompt. Don't enter a pass-phrase; just press ENTER. In the following image, the user has given some command line options to `ssh-keygen`; these are not necessary.

If the command `ssh-keygen` is not found, you may need to install some missing tools. See Section 7.1.1.



To see your public key, enter `cat id_rsa.pub`.

```
> cd .ssh
> cat id_rsa.pub

 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQC4+iMUZvB6VObdDAiOOOaAg16ZMtlJEYDTFite+QSpIF5/5aMcoExJrVLOWa
eAHufDIelwinMHylUAFzdXh+8aIu88aV1dP4RAlTDN+7bwpoE0bQ7cOjURIkUCvuQHD5DY/ToPcrUkAILZCwhAkMjTbiDcvcKa
81OND1Xc7slHeSazy7pwrxiJSGL6CwfdmXjZhZdy01p8gH2+mPha/OTeYaukU/c1vvbuo3GwT1Kx99ruFhY76qbwuRjJlHbb4a
YyCc99v1RIARVUKmAnptAczpcm4zykg9U86PDpUpO8Koh8P7SvzgS9rq3NFWYkgwsF3Kwdn3brwWXvnhN7V5vLmq8bnFWNU7Ka
OvZA43/BbkTbJmol65jTgWjt8UDDJDj7AhFQwXayDG0ksfjfJrbVa/qKsogTvqYgUUpNhd2mWCydzg0MptG4Au2JPxtDS7dLFa
jvOgNVaLSegSXUbtFDqj4gi4/8FuiEN9PqzAazardHqLYAhkThwJnkqCo9H/W4oLeU= john.smith@epita.fr
```

7.2.2 From Linux and MacOS

Open a shell (or `xterm`), and type `cd .ssh ; ls`. If you already have a file named `id_rsa.pub`, then you already have a public key. Otherwise you'll

need to create one with the command `ssh-keygen`. Press ENTER to accept the default value for all prompts. Finally, display the content of the file using `cat id_rsa.pub`. You should see something like the following.

```
> cd .ssh
> cat id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQC4+iMUZvB6VObdDAiOOOaAg16ZMtlJEYDTFite+QSpIF5/5aMcoExJrVLOWa
eAHufDIelwinMHylUAFzdXh+8aIu88aV1dP4RAlTDN+7bwpoE0bQ7cOjURIkUCvuQHD5DY/ToPcrUkAILZCwhAkMjTbiDcvcKa
81OND1Xc7slHeSazy7pwrxiJSGL6CwfdmXjZhZdy01p8gH2+mPha/OTeYaukU/c1vvbuo3GwT1Kx99ruFhY76qbwuRjJlHbb4a
YyCc99v1RIARVUKmAnptAczpcm4zykg9U86PDpUpO8Koh8P7SvzgS9rq3NFWYkgwsF3Kwdn3brwWXvnhN7V5vLmq8bnFWNU7Ka
OvZA43/BbkTbJmol65jTgWjt8UDDJDj7AhFQwXayDG0ksfjfJrbVa/qKsogTvqYgUUpNhd2mWCydzg0MptG4Au2JPxtDS7dLFa
jvOgNVaLSegSXUbtFDqj4gi4/8FuiEN9PqzAazardHqLYAhkThwJnkqCo9H/W4oLeU= john.smith@epita.fr
```

7.2.3 Copy Public Key to Mouse Clipboard

Once the public key has been generated you'll need to copy the key to the mouse clipboard so you can paste it in Section 7.3 and in Section 7.4.

You may simply use `cat` to output the content of the file then select and copy with the mouse.

Listing 7.1 (*Output Key and Copy with Mouse*)

```
1 cmd> cat ~/.ssh/id_rsa.pub
```

MacOS users may select the text with the mouse and press **CMD-C**. Linux and Windows users may select with **CTL-C**.

You may also automate the copy-to-clipboard operation, but it is done differently on each operating system.

Listing 7.2 (*Windows*)

```
1 cmd> cat ~/.ssh/id_rsa.pub | clip
```

Listing 7.3 (*MacOS*)

```
1 cmd> cat ~/.ssh/id_rsa.pub | tr -d '\n' | pbcopy
```

Listing 7.4 (*Linux*)

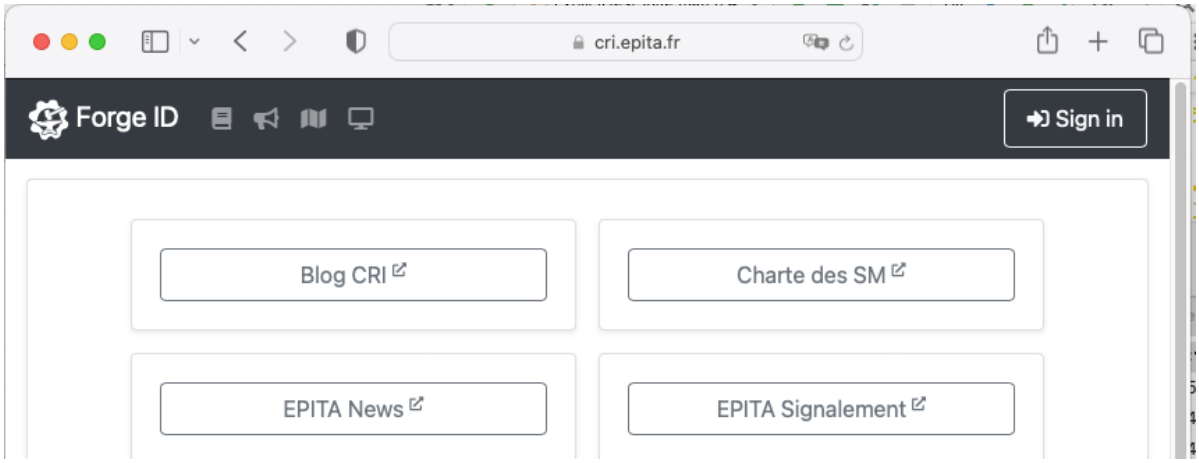
```
1 cmd> cat ~/.ssh/id_rsa.pub | tr -d '\n' \
2                            | xclip -selection clipboard
```

7.3 Register Public Key with Forge

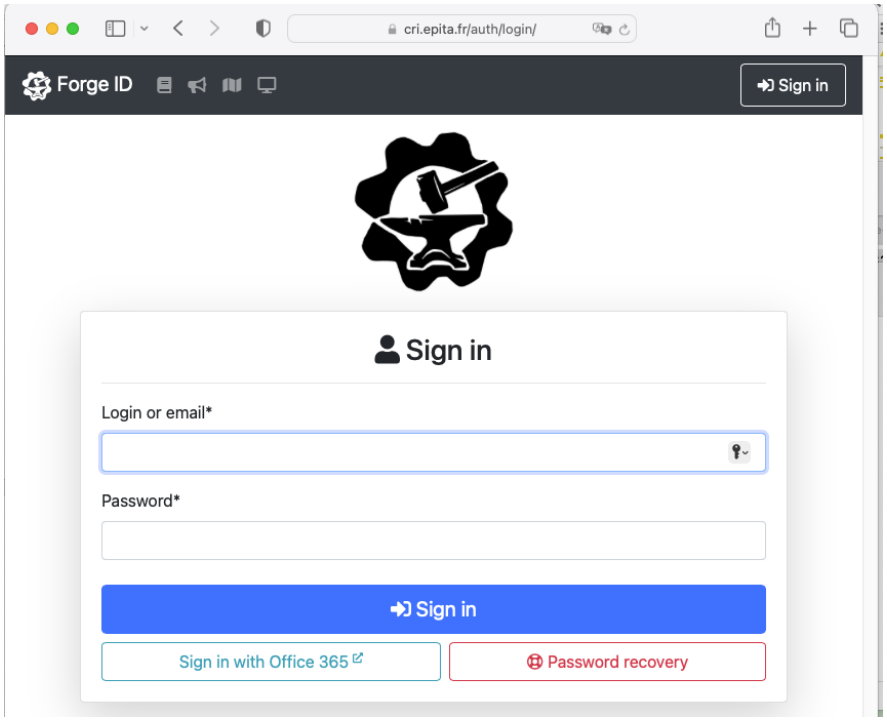
To submit your homework assignments for grading, you'll use the Forge/Intranet services. EPITA students often refer to this service simply as *intra*.

Once you have copied your SSH public key into the mouse clipboard as shown in Section 7.2.3, you must register the key with Forge.

1. Use the URL `https://cri.epita.fr` to sign in to the Forge services.

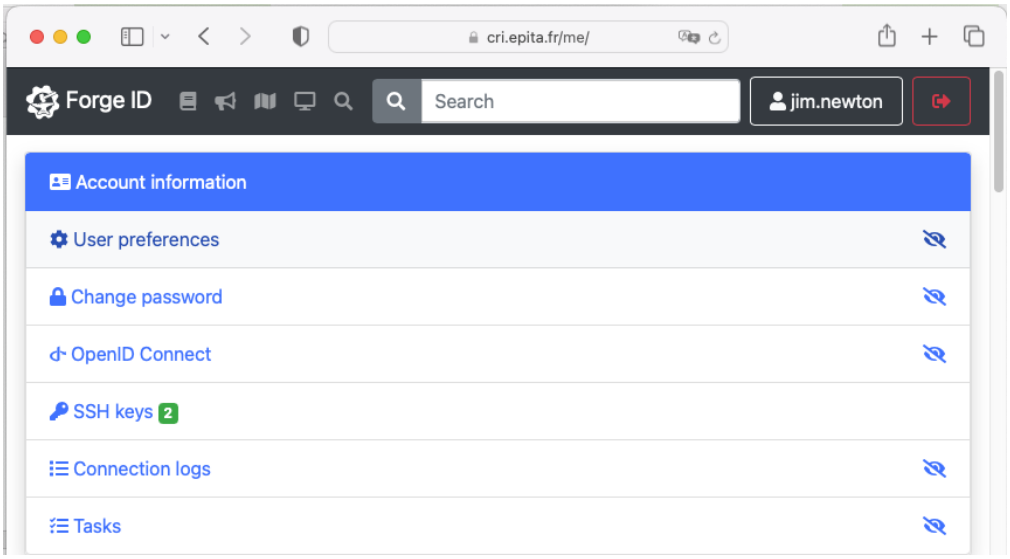


2. Click the **Sign in** link.



3. Enter your EPITA credentials. Your EPITA credentials are usually your first and last name separated by a dot such as `pierre.dupont`, not

`pierre.dupont@epita.fr`.



4. In the Account information form, select **SSH keys** 

.

5. You should now see the following form (with your name) at the bottom of the web page.

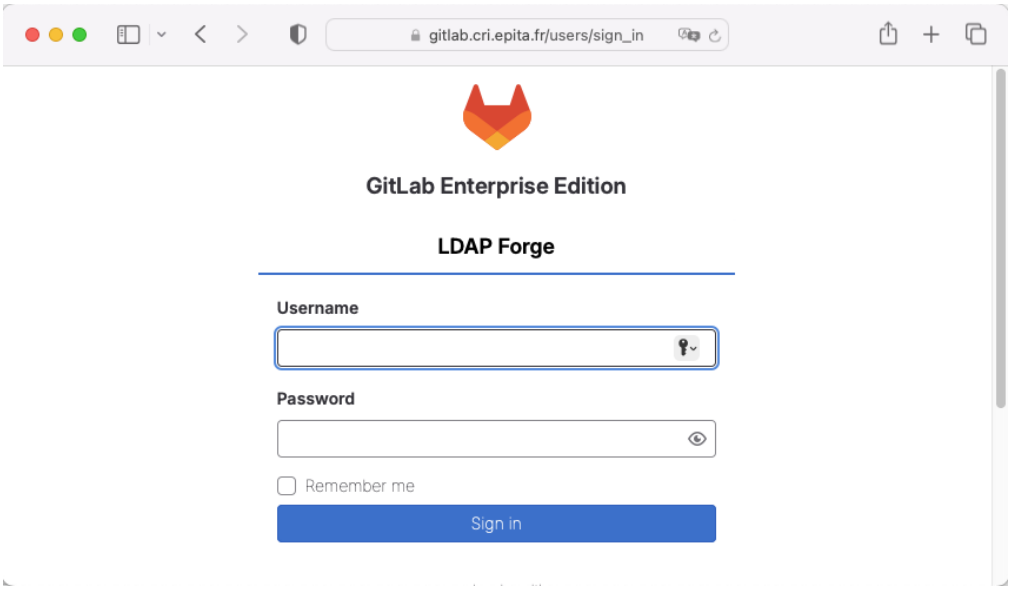
6. Find the type-in field marked **Key*** and paste your public key from the

mouse clipboard (by pressing **CTL-V**, or **CMD-V** for MacOS). Recall that you copied the public key to the clipboard in Section 7.2.3; it is the content of the file `.ssh/id_rsa.pub`.

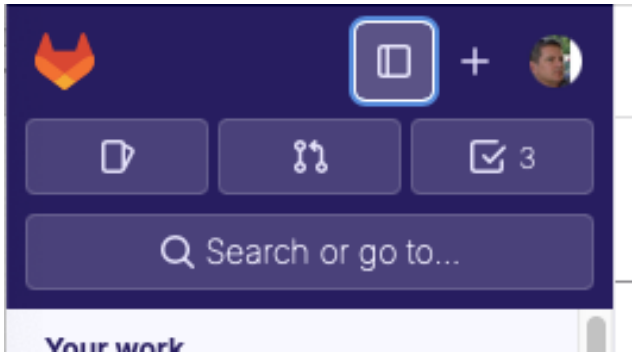
7. Then press the link **Add SSH key** once: 

. Sometimes this is slow—don't press twice. There is no need to enter a **Title**; just leave it blank.

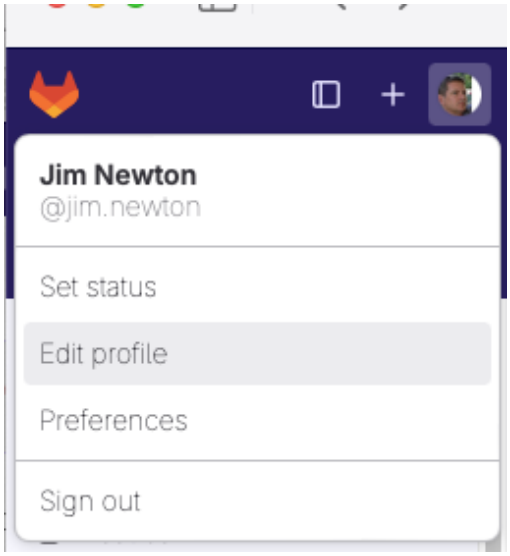
7.4 Register Public Key with GitLab



1. Log into GitLab `https://gitlab.cri.epita.fr` using your EPITA login credentials.

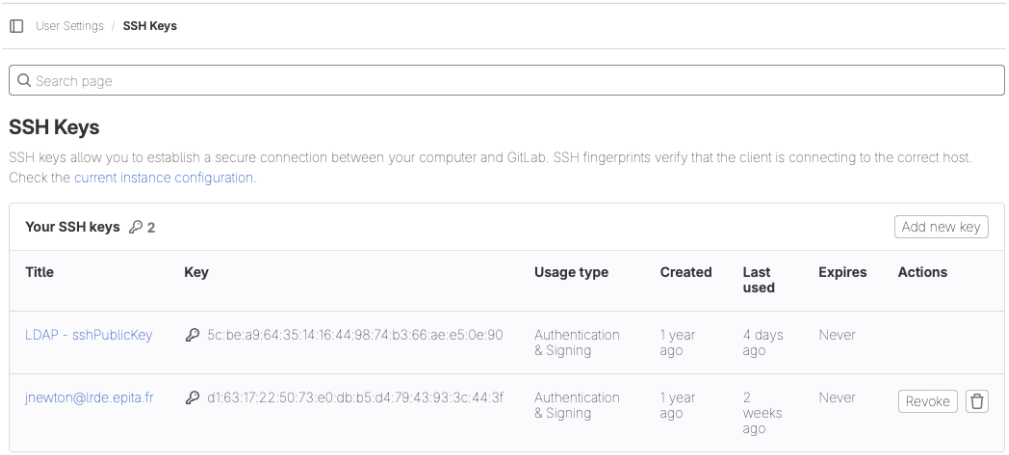


2. Click on your avatar and select **Edit Profile**.



3. Now in the left-hand side, find 

SSH Keys, and click the link, to see a page such as this. You may not yet have any keys listed.

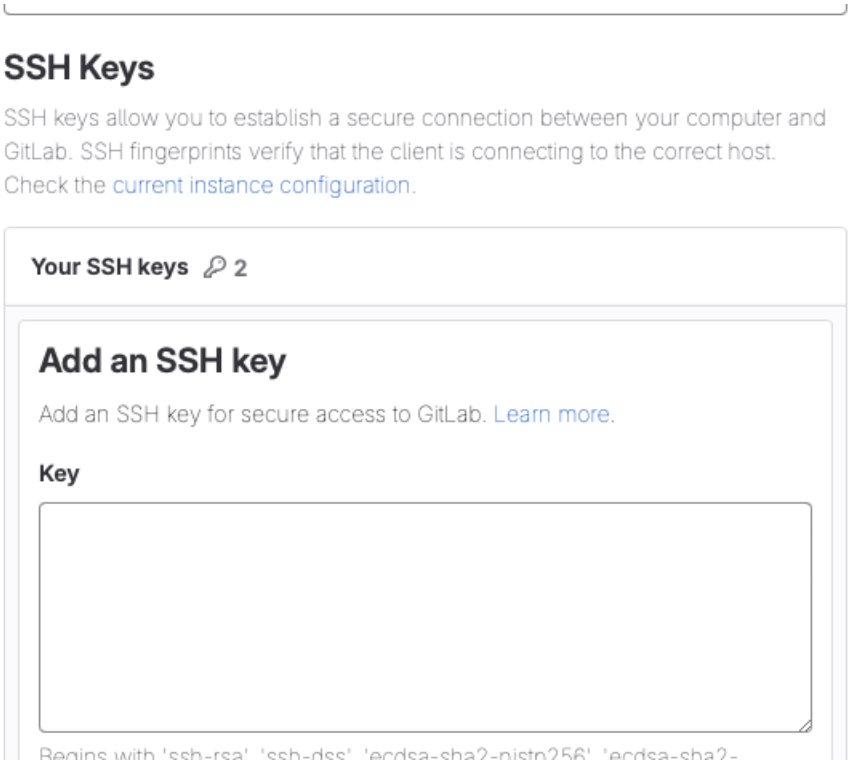


4. Click 

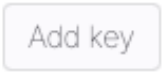
Add new key.

5. You will see a field labeled **Key**. Paste the public key from clipboard into

the **Key** field.



Recall that you copied the public key to the clipboard in Section 7.2.3; it is the content of the file `.ssh/id_rsa.pub`.

6. Finally, scroll down and click the 

Add key button.

7.5 Script `checkout-workarea.py`

Figure 9 illustrates the scripts for unifying and managing the unification of the student repository with the grader repository.

This script assumes a certain structure for the grader repository. In particular it assumes the directory structure of the grader repository (Section 4) and the student repository are exactly the same. The `checkout-workarea.py` populates an empty grader repository with the content (and commit history) of the student repository. The script is written in Python so students can run the script if their laptop is using MacOS, Linux, or Windows. A small number of students don't yet have Python installed on their laptop.

This script is intended for the student to run to checkout the workarea. There is a chicken-and-egg problem. This script is inside the student repository, and the student hasn't yet checked it out. Typically the professor must

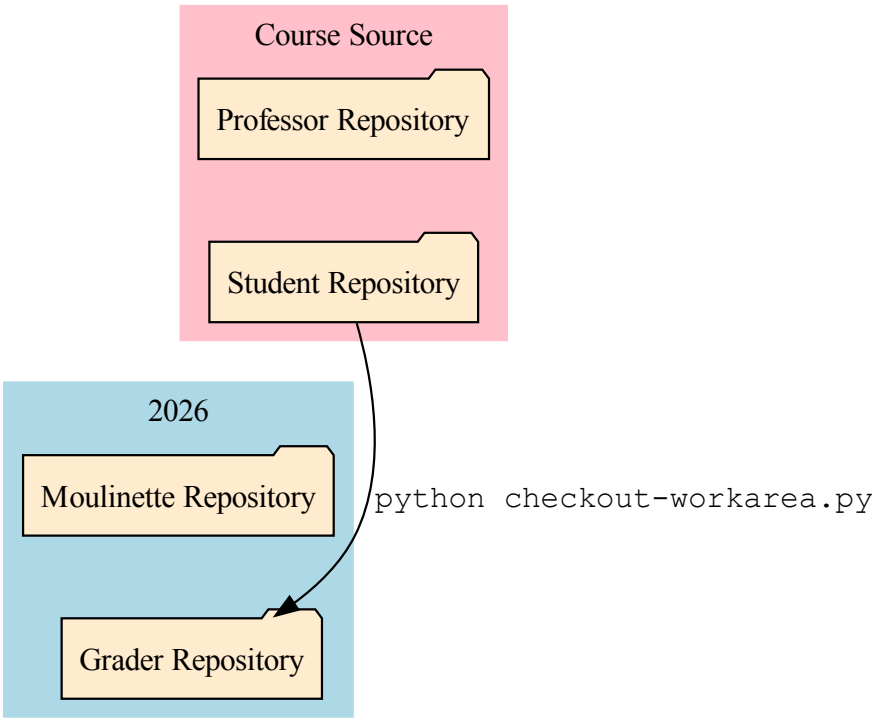


Figure 9: Python Script which initially unifies student repository and grader repository.

distribute this script to the students via Teams chat, or the student uses the web browser to navigate to the student repository in GitLab. Of course the URL is different per course so again the professor will have to distribute the URL to the students. *E.g.*, the page for the Bachelor of Computer Science Linear Algebra course is: https://gitlab.cri.epita.fr/jim.newton/linear-algebra-student. From there the student can navigate to the `bin/` directory and download the script `checkout-workarea.py`. At which point the student can run the script from the shell (or `powershell`) as shown:

```
python checkout-workarea.py
# or perhaps
python3 checkout-workarea.py
```

The `checkout-workarea.py` effectively does the following, except with more extensive error checking to help students who know nothing about `git`.

Listing 7.5 (*Merge Student Repository into Grader Repository*)

```
1  # check out the empty grader repository from forge/intra
2  git clone \
3   git@gitlab.cri.epita.fr:path/to/cri/grader-of-student.git \
4   grader
5
6  cd grader
7
8  # establish remote repository pointing to the
9  #  student repository
10 git remote add student \
11       git@gitlab.cri.epita.fr:path/to/student-repo.git
12
13 # download data from student repository
14 git fetch student
15
16 # merge entire student repository history into
17 #   grader repository, preferring whatever the grader
18 #   repository already has if anything conflicts (same
19 #   strategy as import-to-workarea.py, and for the same
20 #   reason: a second run of this script, e.g. after a
21 #   first attempt that didn't finish cleanly, must never
22 #   overwrite work the student has already committed)
23 git merge --allow-unrelated-histories -X ours master student/master
24
25 # remove remote to avoid confusion in the future
26 git remote remove student
```

`-X ours` resolves an ordinary conflict automatically – both sides added or changed the same path – by keeping the grader repository's version: the same strategy, and for the same reason, as `import-to-workarea.py` (Section 8.4) – a second run of this script, e.g. after a first attempt that didn't finish cleanly, must never overwrite work the student has already committed. It cannot resolve every possible conflict (e.g. one side deleting a path the other side modified); if the merge still fails, the script aborts back to a clean pre-merge state (`git merge -abort`) and stops, rather than risk pushing a partially-merged, conflict-marker-laden result.

The `student` remote above is likewise removed right after use, never left configured – same reasoning as `import-to-workarea.py`'s own temporary remote (Section 8.4 explains why: accidental student pushes and tag pollution).

8 Ongoing Student Experience

Students with prior Forge/Intranet experience will probably find the workflow no different from other courses they have taken, except that the student must occasionally `git pull` the student repository if it has changed. The student repository may change if bugs are fixed in the test cases, additional documentation is added, lecture notes are updated, *etc.*

However, for the courses offered in the International Program (Bachelor of Computer Science) the students have not passed through the EPITA preparatory program, and have never used the Forge/Intranet service prior to coming to EPITA. For this reason the management of the workarea is automated in what is hopefully a foolproof way.

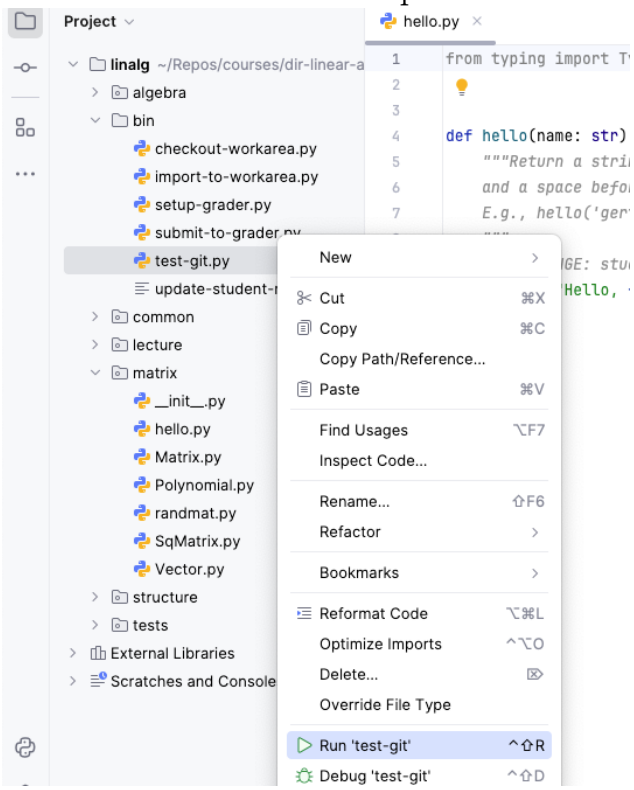
8.1 Script `test-git.py`

The student can run this script to assure that `git` is installed correctly.

The student can run this (as well as `submit-to-grader.py` and `import-to-workarea.py`) from the pop-up menu in the IDE, *e.g.*, PyCharm.

8.2 Script `test-ssh.py`

The student can run this script to assure that the `ssh` keys are set up correctly.



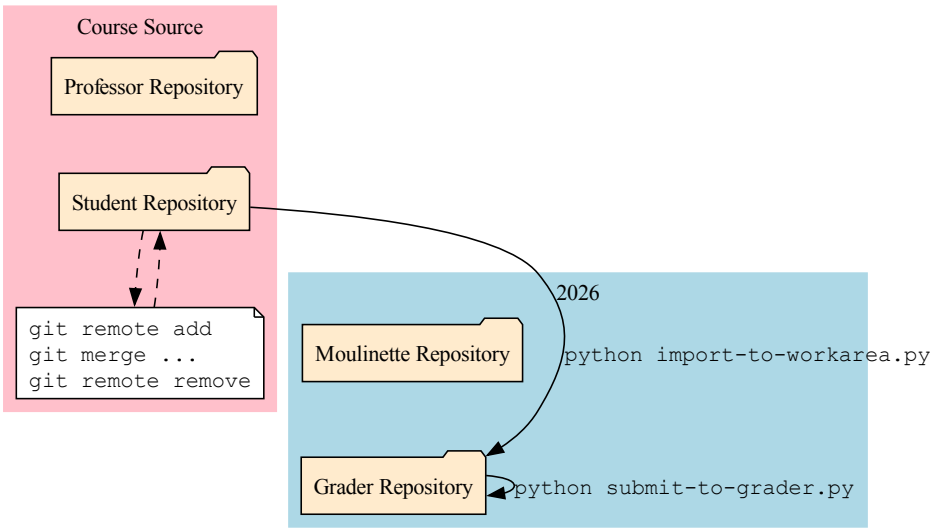


Figure 10: Python Scripts which manage the unified student repository and grader repository.

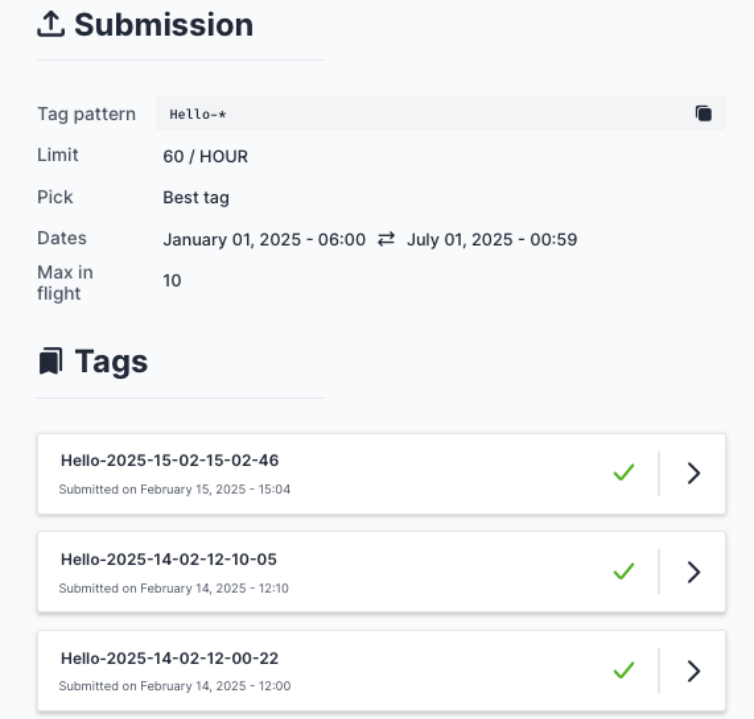
8.3 Script `submit-to-grader.py`

Figure 10 illustrates the use of `submit-to-grader.py`.

Similar to the menu item to run `test-git.py`, the student can also run `submit-to-grader.py` from the menu within the IDE. The script will `git add` and `git commit` all modified files in the workarea. This script will ask the student which assignment to submit. The student must type a prefix such as `Hello`, after which the script will tag the workarea with `Hello-XXXX` where *XXXX* is some unique identifier based on the date and time. Then `git push` and `git push -tags` are executed. Within a few seconds the student will be able to see the results of the auto-grader by viewing the Forge web page.

In the newer version of this script (`calculus` and `algebra-1` as of 2026-09-01), the student does not have to remember or type a tag prefix at all: before prompting, the script fetches `activity.json` live – via `git clone -depth=1` of the Student Repository itself, discarded after reading, and repeated on every run rather than trusted from a cache, since which submissions are open depends on today's date – and lists by name only the assignments currently open for that student's own login (matched against each submission's `subscriptions.logins`, falling back to the assignment-level list for the whole-class-roster pattern), letting the student pick by number. See Section 3.4.4 for

how `activity.json` reaches the Student Repository in the first place – if a student reports seeing no assignments listed as open when one clearly should be, a stale copy there (rather than anything wrong on the student's end) is the first thing to check.



8.4 Script `import-to-workarea.py`

Figure 10 illustrates the use of `import-to-workarea.py`.

The `import-to-workarea.py` can also be run from the IDE menu. The script reestablishes a link to the remote student repository, fetches any new commits on its default branch and then resynchronizes the student repository content into the grader repository. This type of `git merge` operation risks conflicts; therefore, a merge strategy is used to minimize this possibility; in particular, any file modifications in the grader repository are assumed to be precious to the student, so conflicts in the incoming files are ignored.

In some rare cases, the student may desire to merge from the student repository in unsafe mode, in which case the student must do the merge from the command line and must fix any conflicts manually as well.

The link to the Student Repository is deliberately *never* kept as a permanent `git` remote on the grader workarea – it is added just before the fetch/merge above and removed again (`remove_gitlab_remote()`) immediately afterward, every time this script runs. Two independent reasons, both learned the hard way rather than decided up front:

1. **Accidental pushes.** A beginner student, unfamiliar with `git` and landing on the Student Repository first when reading the course materials, can easily try to push a submission there instead of to `origin` (the Grader Repository) if a remote pointing at it is just sitting in their workarea. The Student Repository is read-only to students in the first place, so such a push would fail regardless – but a confusing failure with no obvious next step is still a support burden this avoids entirely by never leaving a remote configured to push to.

2. **Tag pollution.** Forge/Intra rejects the *entire* push to the Grader Repository if even one tag on it doesn't match one of the patterns declared in `activity.yml` (`tagPrefix`) – not just the offending tag (see Section 8.3). `git fetch` follows tags by default, so a remote pointing at the Student Repository that stayed configured across multiple runs of this script risks a later `git push -tags` carrying over a tag that was never meant to reach the Grader Repository at all, and getting the whole push rejected. Keeping the remote strictly temporary is the first layer of defense; the second is stripping tags outright, next.

`import-to-workarea.py` also strips any tag the fetch could have introduced, both right after the merge and again after the resulting push (belt-and-suspenders): `remove_new_tags(baseline_tags)`, where `baseline_tags` is captured *before* the fetch. An earlier version, `remove_existing_tags()`,  deleted every local tag unconditionally instead of only the ones newly introduced by the fetch – harmless the first time this script ever ran anywhere, but it would have wiped an entire grader repository's own tag history (autograder results, past submissions) the first time it ran somewhere that already had any. Fixed 2026-09-02 and propagated to every course using this pattern; if you find an older course whose `import-to-workarea.py` still has the unconditional version, it is safe to apply the same fix.