The SHINE framework

Antoni Marcinek

Institute of Nuclear Physics, Polish Academy of Sciences, Kraków, Poland

SHINE Autumn School, 2 November 2020, Zoom

The SHINE framework

https://gitlab.cern.ch/na61-software/framework/Shine

What it is

- SHINE is a software library and a set of executables for offline data processing in the NA61/SHINE experiment and some online tasks (QA, monitoring).
- online — during data taking, offline — once data is written to storage (CASTOR/EOS)
- It derives from the Pierre Auger Observatory experiment Offline Software and shares some core components with it.

The role of the framework

- defines the event structure and tools to handle it (IO, graphical event browser)
- provides detector description
- provides the set of applications required for common and centralized tasks: reconstruction, MC, QA
- provides utilities for development of more specific tasks: calibration, analysis

Beyond the framework

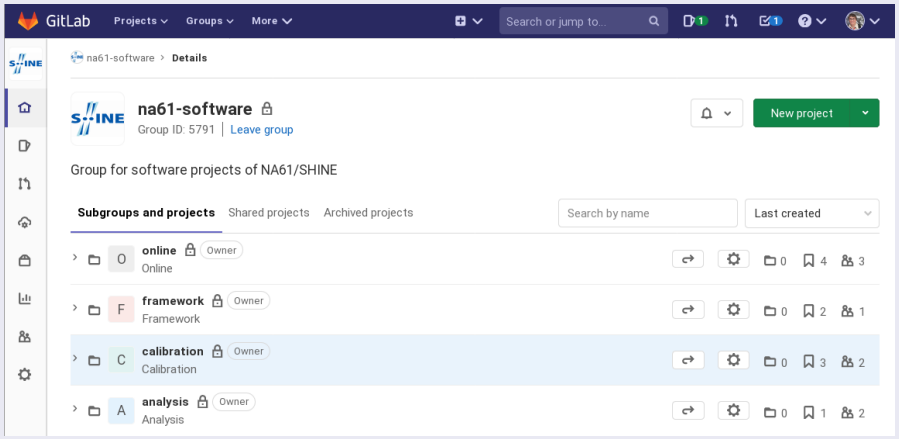
Data base

https://gitlab.cern.ch/na61-database/DB

- holds calibration constants in the form of XML and ASCII files along with XML tables (called global keys) to match a single parameter to versions of constants

Non-framework software

https://gitlab.cern.ch/na61-software



- doesn't use SHINE (online software, legacy calibration)
- uses SHINE but is too specific for the main SHINE project (calibration, analysis)

What you should (already) know

Things which you need to know if you want to use or develop SHINE:

- Linux — we don't support other systems; it is the system of our main production environment at CERN (`lxplus` and `lxbatch`)
  - windows is probably impossible beyond virtual machines
  - macOS might be possible (macOS is unix-based), but you are on your own
- basic Git usage (see next slide)
- intermediate C++ learn it!
  - it is your main tool in the field whether you like it or not
  - there are good free sources of knowledge:
    Thinking in C++ by Bruce Eckel
    `https://www.mindviewllc.com/quicklinks/`
- ROOT toolkit (SHINE still uses ancient ROOT 5, but for your macros you can use ROOT 6 with modern C++ support) read the docs:
  reference manual and the users guide, especially chapters
  - Histograms, Graphics and the Graphical User Interface, Fitting histograms
  - Input/Output, Trees, Object Ownership
  - Math Libraries in ROOT

Git — not only for SHINE, but everything you do

http://www-cs-students.stanford.edu/~blynn/gitmagic
https://git-scm.com/book/en/v2

Proper config

- For every machine you are working on (laptop, desktop, lxplus)
  - Put this as ~/.gitignore_global
  - Execute filling properly your data:

```
git config --global user.name "first.name last.name"
git config --global user.email "your.primary@email.address"
git config --global http.emptyAuth true
git config --global core.excludesfile ~/.gitignore_global
```

- CERN recommends usage of Kerberos with GitLab. Note that some of you have several accounts (e.g. for other experiments) → in this case use SSH protocol to communicate with GitLab (see KB0003137)

Proper usage

- Do development on a new branch: `git checkout -b mynewbranch`
  merge to / rebase onto `master` after `git pull` on `master`

Other useful tools and skills

- good text editor with syntax highlighting (e.g. Vim, Emacs)
  https://www.vim.org/
  http://www.viemu.com/a-why-vi-vim.html
  https://www.gnu.org/software/emacs/
- LaTeX beamer (used to prepare this presentation; useful for iterative presentations of your results when you include loads of pictures)
  https://ctan.org/pkg/beamer
- XML and XML schema
  https://www.w3schools.com/xml/default.asp
  https://www.w3schools.com/xml/schema_intro.asp
- object-oriented design (OOD)
  https://en.wikipedia.org/wiki/Object-oriented_design
  https://en.wikipedia.org/wiki/Design_Patterns
- test-driven development (TDD)
  https://en.wikipedia.org/wiki/Test-driven_development
  https://martinfowler.com/articles/mocksArentStubs.html

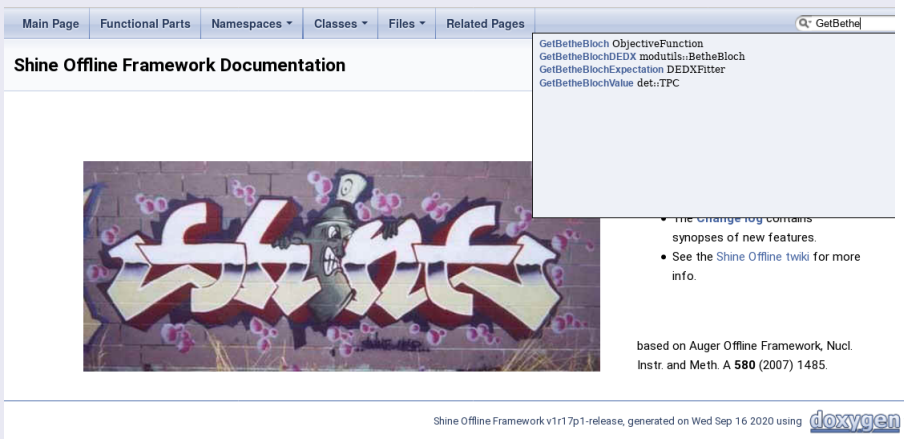
Where to search for documentation

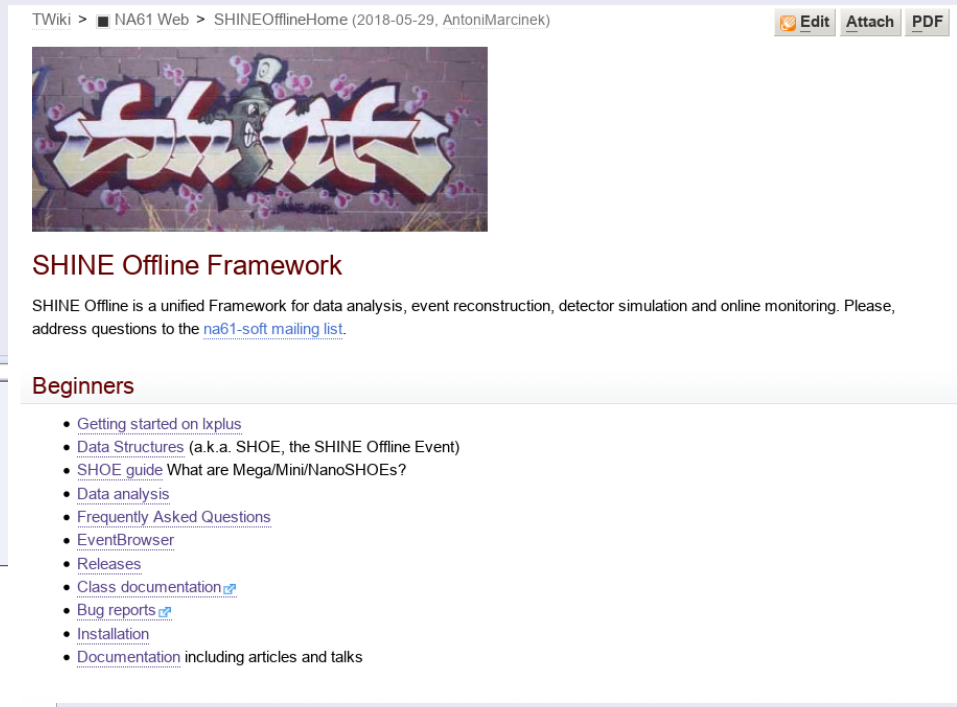
https://twiki.cern.ch/twiki/bin/viewauth/NA61/SHINEOfflineHome
https://shinedoc.web.cern.ch/shinedoc/doxygen/

The main sources of SHINE wisdom

- The SHINE TWiki (not the main TWiki of the experiment) →
- Searchable doxygen-generated documentation

↓





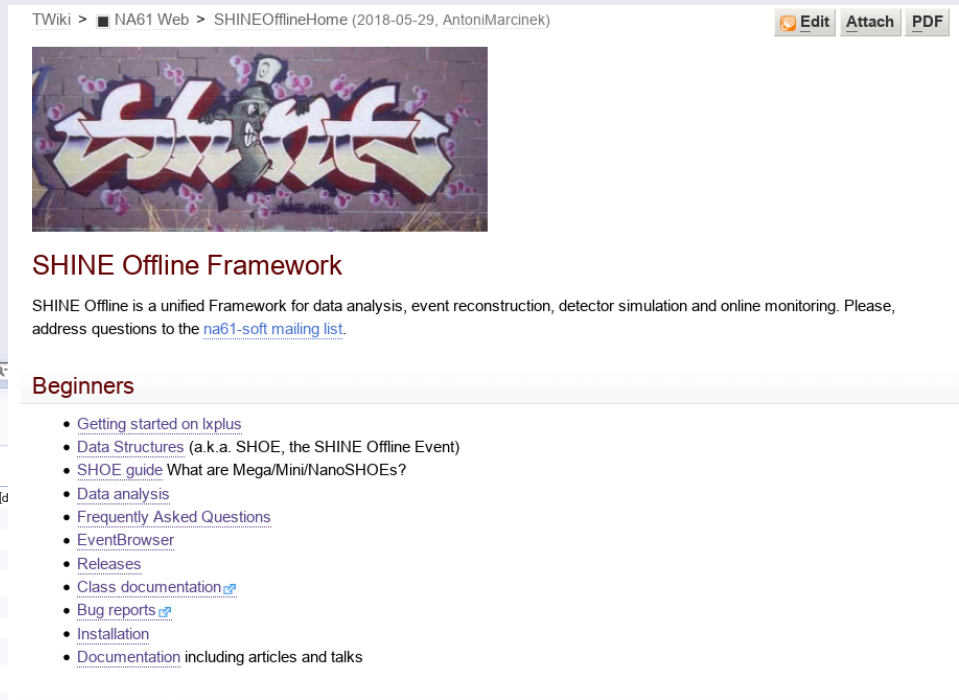
Where to search for documentation

https://twiki.cern.ch/twiki/bin/viewauth/NA61/SHINEOfflineHome
https://shinedoc.web.cern.ch/shinedoc/doxygen/

The main sources of SHINE wisdom

- The SHINE TWiki (not the main TWiki of the experiment) →
- Searchable doxygen-generated documentation (explore it!)

↓



Where to search for documentation cont.

- Releases and the way to set the environment to use SHINE on lxplus and in batch scripts are documented on the TWiki
  https://twiki.cern.ch/twiki/bin/view/NA61/SHINEReleases
- If you have questions, write to the `na61-soft@cern.ch`. We have a JIRA system, but it should be used as a beefed-up TODO list, i.e. no questions, only statements of problems to fix.
- Additional texts are shipped with SHINE itself in the `Documentation/` directory, see subdirectories
  - `DB`
  - `ReferenceManual`
  - `DetectorManagersAndModules`
  - `CalibrationChain`

  although don't start with it, as it is known to be incomplete and/or out-dated (see e.g. [SHINE-366])
- We search for the documentation manager: [SHINE-118]

Structure of the framework

- Explore it from top to bottom. Especially Program Steering and Configuration has very basic information, e.g. how modules and managers obtain their configuration via the `fwk::CentralConfig` from XML files, describes bootstrap.xml file.
- Detector User Interface: start with `det::Detector` class; it is the interface to the detector information for use in modules; gets data through the backend
- Managers — Detector Backend — read detector information from DB and other files on request from the `det::Detector` components
- SHOE — the event structure (see talk by Michael)
- Run — similar description as Event, but valid for all events in the file
- Offline I/O — SHOE files and other formats that SHINE can translate into SHOE

Structure of the framework cont.

- Utilities — for use in modules, managers, external analysis and calibration software
- Modules — process events, steered via XML by the `fwk::RunController`
- Legacy — wrapped NA49 reconstruction software written in C and Fortran, the only reason for 32bit builds

Structure of the sources directory

```
antek@top ~/Install/Shine/sources > ls -lh --color=auto --group-directories-first
total 108K
drwxr-xr-x  5 antek antek 4.0K Apr 24  2020 Applications
drwxr-xr-x  2 antek antek 4.0K Oct 31 00:03 CMakeModules
drwxr-xr-x  2 antek antek 4.0K Jul  7  2018 CMakeTests
drwxr-xr-x  7 antek antek 4.0K Jun  5 06:19 Data
drwxr-xr-x  9 antek antek 4.0K Oct 31 00:03 Documentation
drwxr-xr-x 10 antek antek 4.0K Jun  5 06:19 EventIO
drwxr-xr-x 12 antek antek 4.0K Oct 28 22:29 Framework
drwxr-xr-x 13 antek antek 4.0K Jun  5 06:19 Legacy
drwxr-xr-x  9 antek antek 4.0K Jun  5 06:19 Modules
drwxr-xr-x  8 antek antek 4.0K Jul 11  2018 Tools
drwxr-xr-x 23 antek antek 4.0K Oct 29 13:25 Utilities
-rw-r--r--  1 antek antek  22K Sep 15 23:56 CHANGES
-rw-r--r--  1 antek antek 5.8K Jun  5 06:19 CMakeLists.txt
-rw-r--r--  1 antek antek  114 Sep 27  2017 INSTALL
-rw-r--r--  1 antek antek 4.0K Sep 27  2017 LICENSE.Auger
-rw-r--r--  1 antek antek 2.2K Sep 27  2017 LICENSE.Shine
-rw-r--r--  1 antek antek  300 Jul 11  2018 README
-rw-r--r--  1 antek antek  787 Jul 11  2018 ShineGeneral.dtd.in
-rw-r--r--  1 antek antek  452 Jul 11  2018 TODO
-rwxr-xr-x  1 antek antek 5.4K Oct 31 00:03 shine-offline-config.in
antek@top ~/Install/Shine/sources >
```

- Applications: examples, validation tests; some cleanup is in order [SHINE-119]
- CMakeModules and CMakeTests: components of the build system
- Data: small data files needed in tests
- EventIO: data files handlling
- Framework: steering, detector interface and backend, Event structure
- Tools: executables sources and scripts

Structure of the installation directory

```
antek@top ~/Install/Shine/pro/install > ls -lh --color=auto --group-directories-first
total 48K
drwxr-xr-x  5 antek antek 4.0K Nov  1 17:44 apps
drwxr-xr-x  2 antek antek 4.0K Nov  1 17:44 bin
drwxr-xr-x  5 antek antek  20K Nov  1 17:44 config
drwxr-xr-x  7 antek antek 4.0K Nov  1 17:44 data
drwxr-xr-x  7 antek antek 4.0K Nov  1 17:44 doc
drwxr-xr-x 11 antek antek 4.0K Nov  1 17:44 include
drwxr-xr-x  2 antek antek 4.0K Nov  1 17:44 lib
drwxr-xr-x  6 antek antek 4.0K Nov  1 17:44 scripts
antek@top ~/Install/Shine/pro/install >
```

```
antek@top ~/Install/Shine/pro/install > ls -lh --color bin
total 696K
-rwxr-xr-x 1 antek antek 152K Nov  1 17:41 ShineMerge
-rwxr-xr-x 1 antek antek 201K Nov  1 17:44 ShineOffline
-rwxr-xr-x 1 antek antek 113K Nov  1 17:38 calibMerge
-rwxr-xr-x 1 antek antek 4.0K Jul 11  2018 createShineModuleSkeleton.py
-rwxr-xr-x 1 antek antek  51K Nov  1 17:41 eventBrowser
-rwxr-xr-x 1 antek antek 130K Nov  1 17:38 rootdiff
-rwxr-xr-x 1 antek antek 6.1K Nov  1 17:38 shine-offline-config
-rwxr-xr-x 1 antek antek  25K Nov  1 17:44 shoot
antek@top ~/Install/Shine/pro/install >
```

- apps = Applications
- bin
  - ShineOffline — runs sequence of modules given a `bootstrap.xml` file
  - createShineModuleSkeleton.py — use this to implement your own module
  - eventBrowser — get a SHOE file and see the events (you've already seen plots from the browser)
  - shoot — ROOT linked with SHINE
- config: module and manager XML and XML schema (XSD) files
- data = Data
- doc = Documentation
- include: header files divided according to namespaces (differently than in sources)
- lib: libraries that you can link with your software if you need SHINE elements
- scripts: environment setting on lxplus + other scripts (I don't know myself what they are...)

Managers

- Detector Backend — read detector information from DB and other files on request from the `det::Detector`.
- `det::Detector` learns about the available managers from a DManagerRegister configuration entry, e.g. from the DB:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<DManagerRegisterConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='[SCHEMAPATH]/DManagerRegisterConfig.xsd'>

  <manager> BPDStaticInfoXMLManager </manager>
  <manager> BPDGeometryXMLManager </manager>
  <manager> BPDStripGainXMLManager </manager>
  <manager> BPDStripGainFixedManager </manager>
  <manager> G4GeometryFixedManager </manager>
  <manager> G4MaterialFixedManager </manager>
  <manager> GasBagStatusXMLManager </manager>
  <manager> MHTDCCalibrationXMLManager </manager>
  <manager> MHTDCCalibrationFixedManager </manager>
  <!-- Uncomment when residuals are ready for any 2018 production
  <manager> TPCResidualsXMLManager </manager>
  -->
```

- Managers work in a chain of responsibility, so if `BPDStripGainXMLManager` provides BPD strip gains for a given run number/time stamp, the `BPDStripGainFixedManager` is ignored.

Managers cont.

- Different types of managers:
  - Static: represent things that we know for sure after given data set is taken, e.g. cabling; xml files in SHINE repository
  - Fixed: provide dummy values of calibration parameters for tests; xml files in SHINE repository
  - XML, ASCII, Text, etc.: provide proper values of calibration parameters; files in the DB because calibration constants are obtained iteratively and may have different versions for the same time stamp
- Laziness: `det::Detector` asks only for properties that the user wants, so if the user code never cares about TPCs, no information about TPCs is read from DB. There is some granularity to this, so e.g. asking for any TPC value causes all of them to be read.
- Caching: `det::Detector` caches values with validity run number/time ranges provided by managers, so doesn't ask for everything in every event — updates only invalid information; most of properties (the largest ones) are read only once per data file processing.
- Easy to test: managers only read data from files, so tests need to check few data entries and the overall logic (e.g. if validity ranges are properly treated); test through the detector interface!

Modules

- Modules — process events, steered via XML by the `fwk::RunController`
- Every module needs to implement only 3 methods:
  - `Init`: parses XML config
  - `Process`: does the actual event processing
  - `Finish`: non-necessary for majority of modules, e.g. write out files

  Start implementation running the `createShineModuleSkeleton.py` script.
- `fwk::RunController` learns about modules to run from ModuleSequence.xml file (see next page).
- Difficult to test: it is rather complicated to prepare the input (e.g. clusters for a tracking module) and the expected values are usually not know. Therefore unit tests can check the overall logic, e.g.:
  - `Modules/..../testBPDReconstructorSG.cc`
  - `Modules/..../testTOFPotentialHitFinderSG.cc`

  while most of the functionality is checked in integration tests running on real data and comparing contents of the resulting event(s) to reference values, e.g.:
  - `Applications/IntegrationTests/..../TPCDEDXCalculatorSG`
  - `Applications/Standard/Validation/Reconstruction`

Module sequence

ModuleSequence.xml 4.52 KB

```
<!-- ModuleSequence for reconstruction with legacy clients -->
<sequenceFile
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='[SCHEMAPATH]/ModuleSequence.xsd'>

  <enableTiming/>

  <moduleControl>

    <loop numTimes="unbounded">

      <module>  DSHACKServerLauncherSG </module>
      <module>  EventFileReaderSG      </module>
      <module>  ClientInitializerSG    </module>
      <module>  BPDReconstructorSG     </module>
      <module>  BPDVertexSetterSG      </module>
      <module>  BPDToNA49SG            </module>
      <try>
        <module config="VTPCs">   Dipt256NewModuleSG     </module>
        <module config="MTPCs">   Dipt256NewModuleSG     </module>
        <module config="GPC">     Dipt256NewModuleSG     </module>
        <module config="VT1">     EdistoModuleSG         </module>
        <module config="VT1">     VtNcalcModuleSG        </module>
        <module config="VT2">     EdistoModuleSG         </module>
        <module config="VT2">     VtNcalcModuleSG        </module>
```

Utilities: units

- SHINE pays attention to units, neither framework nor user code should make any assumptions on the units of quantities that it uses.
- If a value is read from an external library or a file, or it is assigned as a constant in the code, multiply by a unit:

```
// Mass() returns GeV
const double mass = TDatabasePDG::Instance()->GetParticle("pi+")->Mass() * utl::GeV;

// The ASCII file assumes GeV
double pCut;
cutsFile >> pCut;
pCut *= utl::GeV;

const double ptCut = 100 * utl::MeV;
```

- If a value is written to screen, file, etc., divide by a unit:

```
// prints "The pion mass is: 139.57 MeV"
cout << "The pion mass is: " << mass / utl::MeV << " MeV" << endl;

TH1D hPt("hPt", ";p_{T} [GeV];counts", 100, 0, 2);
(...)
if (pt > ptCut && p > pCut) // here we don't need to care about the units
  hPt.Fill(pt / utl::GeV); // here we need to
```

Utilities: units and the XML configuration

- Our XML reader handles units for us.
- XML file:

```xml
<?xml version="1.0" encoding="iso-8859-1"?>

<TargetConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="[SCHEMAPATH]/TargetConfig_DBFormat.xsd">

  <materialDensity unit="g/cm3"> 11.34 </materialDensity>
  <materialInteractionLength unit="g/cm2"> 199.6 </materialInteractionLength>
  <materialMolarMass unit="g/mol"> 207.2 </materialMolarMass>
  <centerX unit="cm"> 0 </centerX>
  <centerY unit="cm"> 0 </centerY>
  <centerZ unit="cm"> -602.3 </centerZ>
</TargetConfig>
```

- C++ code:

```cpp
// get data from XML; targetConfig is utl::Branch
const double centerZ = targetConfig.GetChild("centerZ").Get<double>();
const double density = targetConfig.GetChild("materialDensity").Get<double>();

// prints "target position is -6.023 meters"
cout << "target position is " << centerZ / utl::m << " meters" << endl;

// Units support arithmetic
// prints "target density is 11340 kg/m3"
cout << "target density is " << density / (utl::kg / utl::m3) << " kg/m3" << endl;
```

Contributing

- Please read and follow the coding conventions (now easier with clang-format)!
- The things to be done should be described and discussed using JIRA: `https://its.cern.ch/jira/projects/SHINE`
  - Don't use it to ask questions! Send e-mails to `na61-soft@cern.ch`.
  - If you want an issue to be solved in the coming release, set Fix Version to next.
  - It doesn't mean I will solve it, it means I will not make a release until it is solved.
- Don't use `cout`/`cerr`, but SHINE'y ways to communicate information to users: `WARNING`, `ERROR`, `INFO` with conditions on the `ErrorLogger`'s verbosity. [SHINE-SHINE-392]
- Learn about SHINE units.
- Learn to use git properly — next page
- If you push commits to SHINE, GitLab runs builds and tests in an environment matching `lxplus`. Expect e-mail in case it fails.

Contributing — git with GitLab

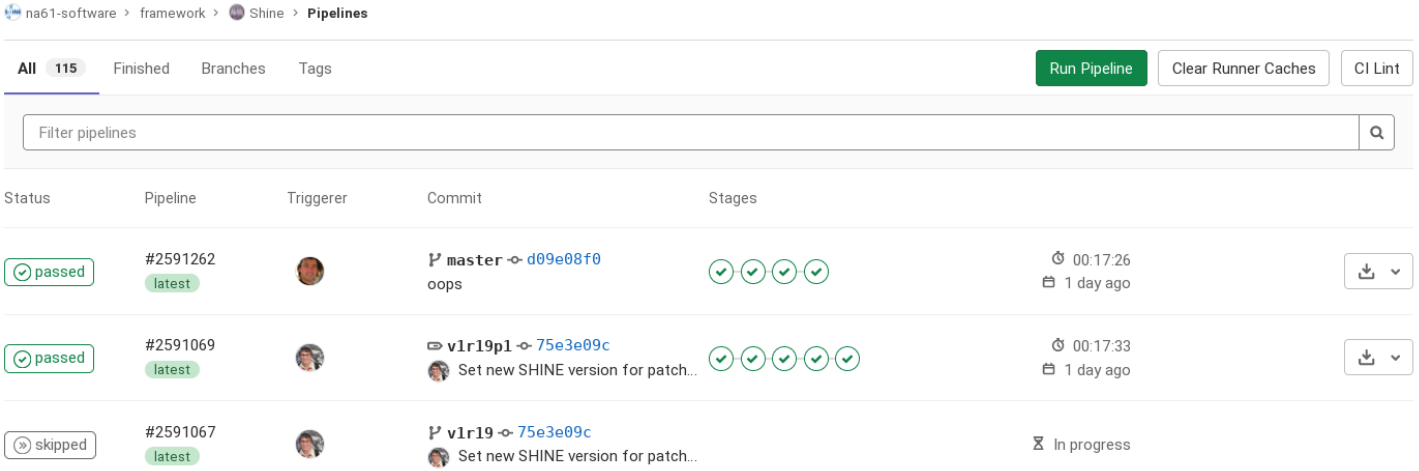
- On every push an e-mail is sent to `na61-soft@cern.ch` summarizing the commits included in the push
- If you include something like ‘see SHINE-100’ in the commit message, then GitLab will create a comment in the respective JIRA issue linking to the commit.
- If you write something like ‘closes SHINE-100’, then GitLab will tell JIRA that the respective issue should be closed.
- After some experimenting I find the following as the best way to reference JIRA issues in git commit messages:

```
[SHINE-392] Add new verbosity level eProgress and make it default

Closes SHINE-392

Note that we don't advertise the 'progress' level in modules' xml files
as it doesn't make much sense to set it given what is its intent: for
regular modules it should be consistent with 'silent'.
```

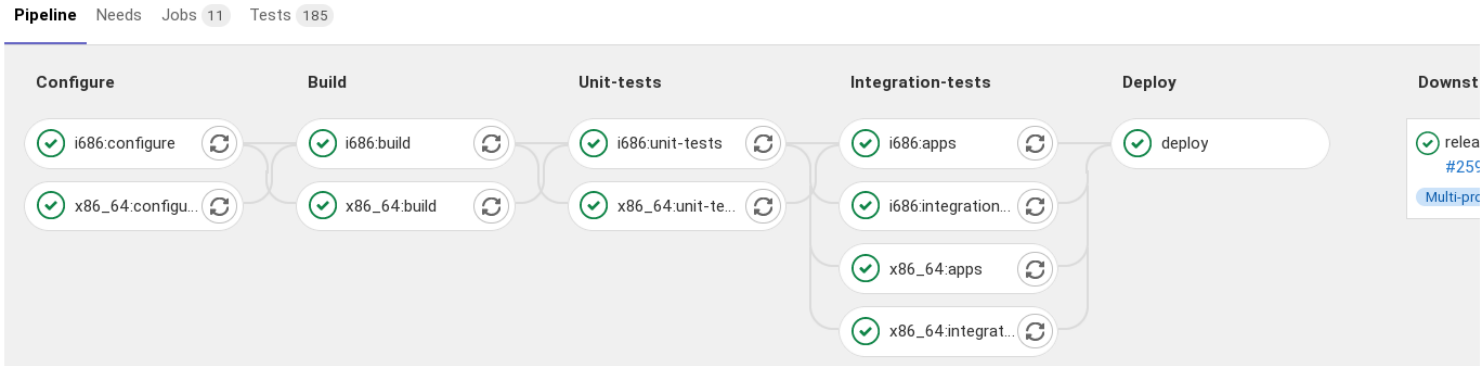
- See how to write good commit messages.
- Avoid single work + merge commits. Have merge commits with several work commits and several remote commits.
- Use graphical tool like `gitk` to look at the sequence of commits before pushing them, especially after `git rebase`.

GitLab CI — pipelines

- CI setup is versioned along with SHINE and visible to every developer
- Dedicated nodes are based on lxplus setup using `Puppet`
- All branches are checked
- Pipeline is divided into stages making it easier to understand what fails:



- Pipeline consists of many jobs, some running in parallel:



- Easy to extend, jobs conditional, pipelines can trigger downstream pipelines

GitLab CI — tests

- Test summaries are browsable from the pipeline view:

Pipeline Needs Jobs 11 **Tests 185**

Summary

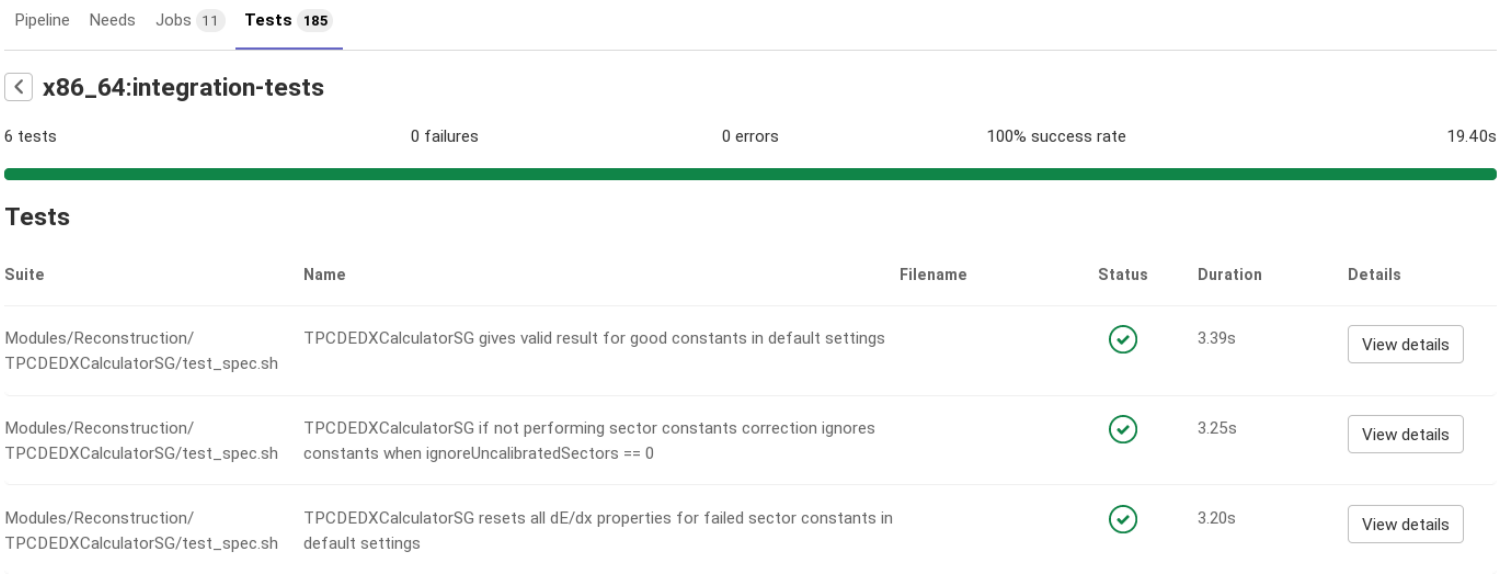
185 tests | 0 failures | 0 errors | 100% success rate | 884.38s

Jobs

Job	Duration	Failed	Errors	Skipped	Passed	Total
x86_64:integration-tests	19.40s	0	0	0	6	6
i686:integration-tests	18.82s	0	0	0	6	6
x86_64:unit-tests	431.74s	0	0	0	87	87
i686:unit-tests	414.42s	0	0	0	86	86

GitLab CI — tests

- Test summaries are browsable from the pipeline view:



GitLab CI — tests

- Test summaries are browsable from the pipeline view:

Pipeline Needs Jobs 11 **Tests 185**

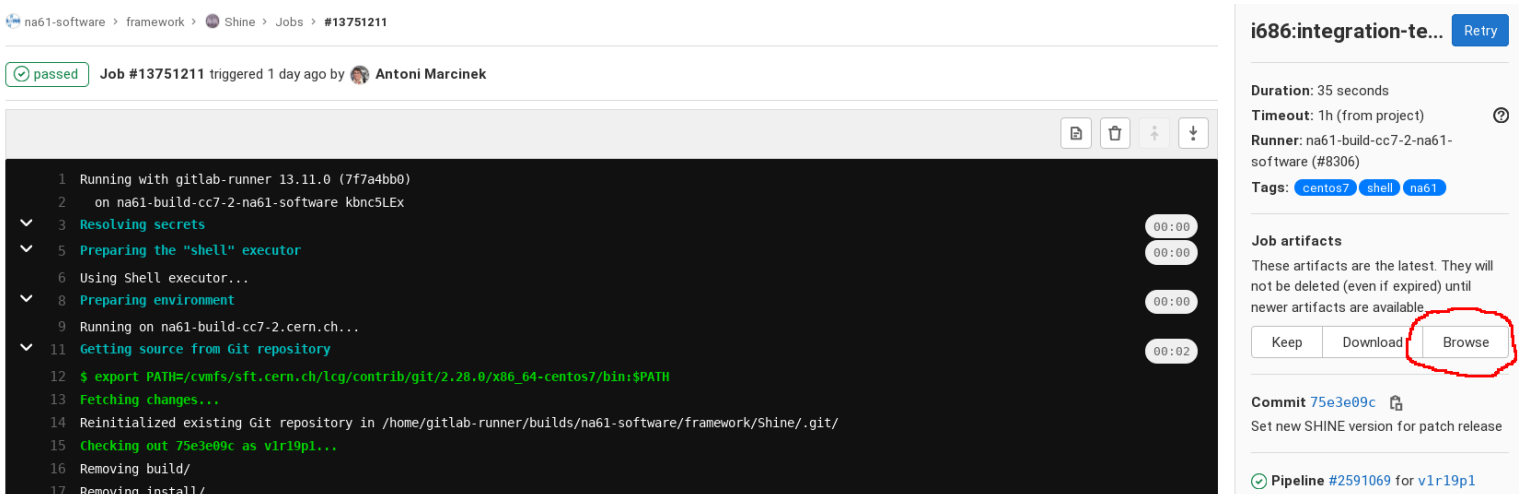
x86_64:integration-tests

6 tests | 0 failures | 0 errors | 100% success rate | 19.40s

Tests

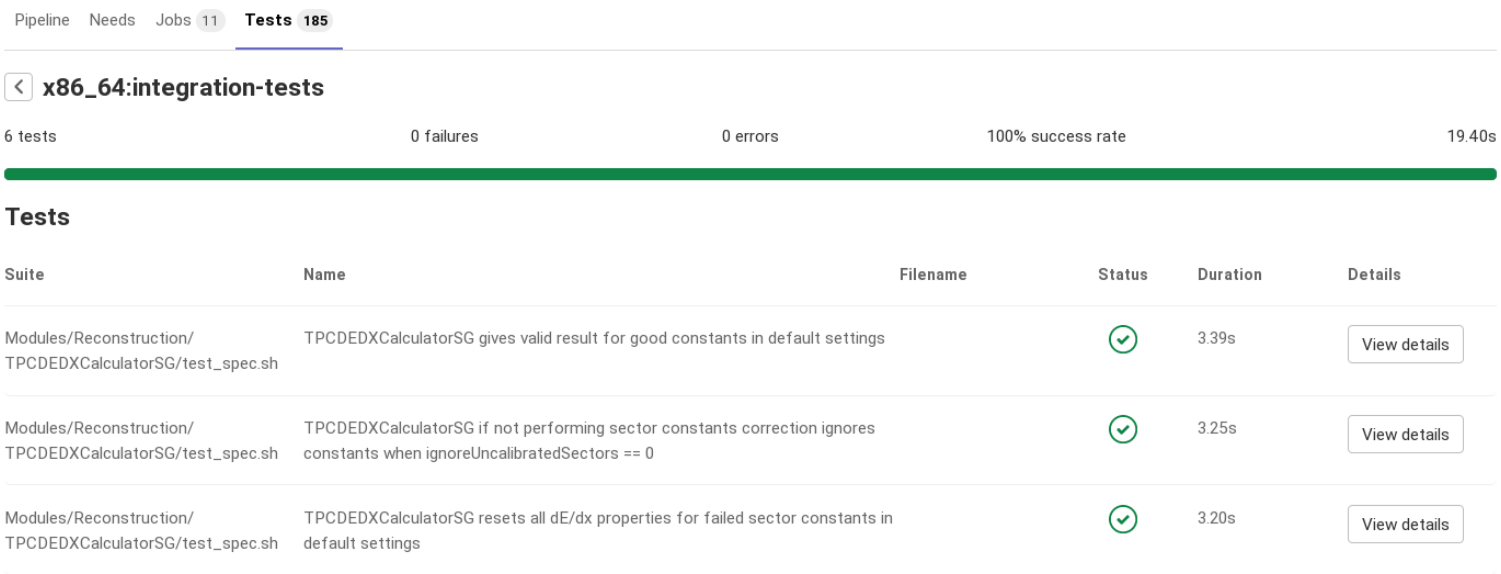
Suite	Name	Filename	Status	Duration	Details
Modules/Reconstruction/ TPCDEDXCalculatorSG/test_spec.sh	TPCDEDXCalculatorSG gives valid result for good constants in default settings		✓	3.39s	View details
Modules/Reconstruction/ TPCDEDXCalculatorSG/test_spec.sh	TPCDEDXCalculatorSG if not performing sector constants correction ignores constants when ignoreUncalibratedSectors == 0		✓	3.25s	View details
Modules/Reconstruction/ TPCDEDXCalculatorSG/test_spec.sh	TPCDEDXCalculatorSG resets all dE/dx properties for failed sector constants in default settings		✓	3.20s	View details

- Detailed reports available as xml files:



GitLab CI — tests

- Test summaries are browsable from the pipeline view:



- Detailed reports available as xml files: