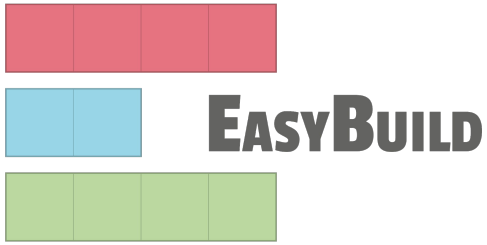


Introduction to EasyBuild

Slides adapted from EESSI webinar, 19 May 2025

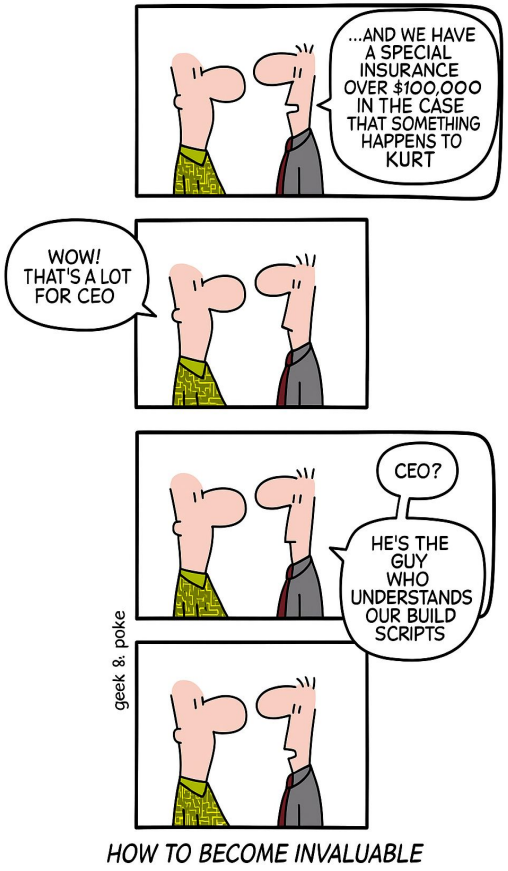
https://eessi.io/docs/training/2025/webinar-series-2025Q2

Kenneth Hoste (HPC-UGent, BE)

Caspar van Leeuwen (SURF, NL)

PART I: Introduction to EasyBuild

Why are we here?



- Had a course in 2021 and 2022, but the number of people in LUST very familiar with EasyBuild has become small since Peter and Orian left.
- And even fewer people are aware of the organisation of everything.
- So hope for more contributions as the load is increasing
  - More users want variants of software
  - Modules built around containers to extend them and make them easier to use
  - Visualisation stack, profiling tools, ...

We need more of those in LUST...

What is EasyBuild?

What is EasyBuild?

- **EasyBuild is a software build and installation framework**
- Strong focus on scientific software, performance, and HPC systems
- Open source (GPLv2), implemented in Python
- Brief history:
  - Created in-house at HPC-UGent in 2008
  - First released publicly in Apr'12 (version 0.5)
  - EasyBuild 1.0.0 released in Nov'12 (during SC12)
  - Worldwide community has grown around it since then! (>1,000 members on EasyBuild Slack)

https://easybuild.io

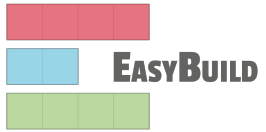
https://docs.easybuild.io

https://blog.easybuild.io

https://github.com/easybuilders

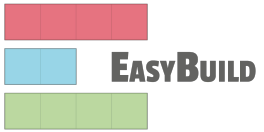
https://easybuild.io/join-slack

EasyBuild in a nutshell



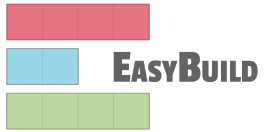
- **Tool** to provide a ***consistent and well performing*** scientific software stack
- Uniform interface for installing scientific software on HPC systems
- Saves time by ***automating*** tedious, boring and repetitive tasks
- Can empower scientific researchers to self-manage their software stack
- **A platform for collaboration among HPC sites worldwide**
- Has become an “expert system” for installing scientific software

Key features of EasyBuild (1/2)



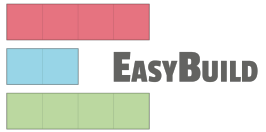
- Supports fully **autonomously** installing (scientific) software, including dependencies, generating environment module files, ...
- **No admin privileges are required** (only write permission to installation prefix)
- **Highly configurable**, easy to extend, support for hooks, easy customisation
- Detailed logging, fully transparent via support for “dry runs” and trace mode
- Support for using custom module naming schemes (incl. hierarchical)

Key features of EasyBuild (2/2)



- Integrates with various other tools (Lmod, Singularity, FPM, Slurm, GC3Pie, …)
- **Actively developed and supported by worldwide community**
- **Frequent stable releases** since 2012 (every 6 - 8 weeks)
- **Comprehensive testing**: unit tests, testing contributions, regression testing
- **Various support channels** (mailing list, Slack, conf calls) + yearly user meetings

Focus points in EasyBuild



Performance

- Strong preference for building software from source
- Software is optimized for the processor architecture of build host (by default)

Reproducibility

- Compiler, libraries, and required dependencies are mostly controlled by EasyBuild
- Fixed software versions for compiler, libraries, (build) dependencies, ...

Community effort

- Development is highly driven by EasyBuild community
- Lots of active contributors, integration with GitHub to facilitate contributions

What EasyBuild is *not*

- EasyBuild is **not YABT (Yet Another Build Tool)**
  - It does not try to replace `CMake`, `make`, `pip`, etc.
  - It wraps around those tools and automates installation procedures
- EasyBuild does **not replace traditional Linux package managers** (`yum`, `dnf`, `apt`, ...)
  - You should still install some software via OS package manager
  - Anything that is run with admin privileges and should be updated in-place (OpenSSL, Slurm, etc.)
- EasyBuild is **not a magic solution** to all your (software installation) problems
  - You may still run into compiler errors (unless somebody worked around it already)

Lmod

Lmod

- See the notes
- Use a fairly standard configuration of Lmod on LUMI
- We do make use of a hierarchy, but not fully
  - Should have had a compiler level too
- Concepts:
  - MODULEPATH environment variable is crucial
  - "One name rule": No two modules with the same name
  - Family concept: No two modules of the same family (but fortunately a bug is present)
  - Module extensions: Map nicely on extensions in EasyBuild, but can be used for Bundles also.

Lmod: Finding modules

- `module spider` command
- `module whatis` : Somewhat problematic at the moment
- `module help` : Information built from parameters in the easyconfig
- `module keyword` : A bit tricky
- And we have the search function in the LUMI Software Library as you can only search for installed modules of course

- Implementation details in the course notes
  - Includes some warnings for things that can go wrong if you manually write modules. E.g., no guarantee about the order of unloading modules
  - Should realise that module unload executes the module file also but reverses some operations!

The HPE Cray Programming Environment

Components

- Universal wrapper, same for all compilers
  - But options are compiler-specific hence different
- Compilers:
  - Cray: cce module
  - GNU: gcc and gcc-native modules
  - AMD AOCC for CPU: aocc module
  - AMD ROCm compilers: amd module
- Message passing toolkit with cray-mpich and OpenSHMEM
- Scientific libraries: cray-libsci, cray-libsci_acc, fftw and various configurations of HDF5 and NetCDF
- Debugger, performance analysis tools, …

Configuring through modules

- PrgEnv modules: system-defined behaviour
- Wrapper configured based on modules loaded and some compiler flags (`-fopenmp`)
  - MPI module, libsci, fftw, …
  - Network target modules: But a bad implementation
  - CPU target module to automatically select CPU optimisation flag.
  - GPU target module: A pain as it automatically enables OpenMP offload when `-fopenmp` is used, and this sometimes causes issues
- There are command line alternatives but our current toolchain implementations follow the module approach and are based on code developed at CSCS

LUMI software stacks

Stacks

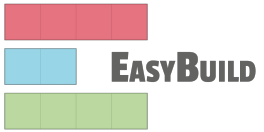
- **CrayEnv**: Like the default login environment, with some packages installed with EasyBuild and management of the target modules
- **LUMI**: Our main software stack built with EasyBuild
  - Versioning corresponding to the versioning of the CPE
  - Within each version, 4 options with different settings for CPU and GPU optimisations: zen2, zen3, zen3+MI250x, zen2+A40 via partition modules
  - Done as a hierarchy, but no level for the compiler toolchain
- **spack**: Want to offer a configuration but no longer a full stack
  - Not indexed in the user cache by default
- Some modules to enable local stacks: **Local-CSC** and **Local-quantum**
  - Not indexed in the user cache by default

Stacks: extra partitions

- Extra partitions just to configure EasyBuild for special tasks:
  - partition/common: Some software for all 4 regular partitions for which zen3-specific optimisations are not important
  - partition/container: Containers that do not really depend on a version of the CPE
  - partition/system: Software available everywhere, e.g., Vampir
  - partition/CrayEnv: Install software for the CrayEnv stack. All software should be independent of any of the CPE modules, so use the compilers that are simply available on the system

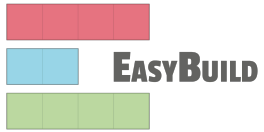
Terminology

EasyBuild terminology



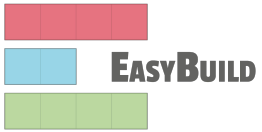
- It is important to briefly explain some terminology often used in EasyBuild
- Some concepts are specific to EasyBuild: easyblocks, easyconfigs, …
- Overloaded terms are clarified: modules, extensions, toolchains, …

EasyBuild terminology speed run: framework



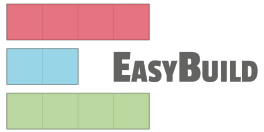
- The EasyBuild framework is the **core of EasyBuild**
- **Collection of Python modules**, organised in packages
- Implements **common functionality** for building and installing software
- Defines abstract installation procedure, in steps (configure, build, test, install, ...)
- Support for applying patches, running commands, generating module files, ...
- Examples: `easybuild.toolchains`, `easybuild.tools`, ...
- Provides `eb` command, but can also be leveraged as a Python library
- GitHub repository: https://github.com/easybuilders/easybuild-framework

EasyBuild terminology speed run: easyblock



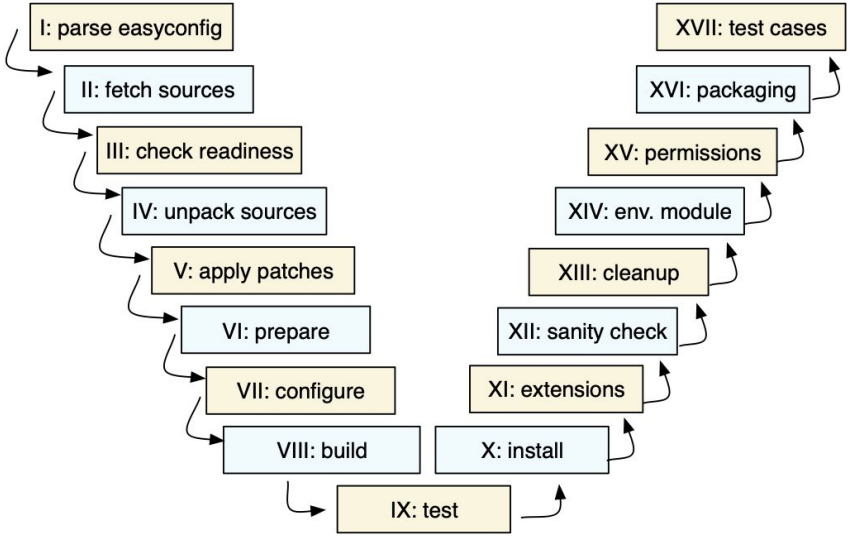
- A **Python module** that implements steps of installation procedure (as defined by framework)
  - Can be viewed as a “plugin” to the EasyBuild framework
- **Generic easyblocks** for “standard” stuff: `cmake` + `make` + `make install`, Python packages, etc.
- **Software-specific easyblocks** for complex software (OpenFOAM, TensorFlow, WRF, …)
- Installation procedure can be controlled via easyconfig parameters
  - Additional configure options, commands to run before/after build or install command, …
  - Generic easyblock + handful of defined easyconfig parameters is sufficient to install a lot of software
- GitHub repository: https://github.com/easybuilders/easybuild-easyblocks
- Easyblocks do not need to be part of the EasyBuild installation (see `--include-easyblocks`)

EasyBuild terminology speed run: easyconfig file



- “Build recipe”
- Text file that specifies what EasyBuild should install (in Python syntax)
- **Collection of values for easyconfig parameters** (key-value definitions), no logic (cfr. easyblock)
- Also specifies which easyblock to use (directly, or indirectly via software name)
- Filename typically ends in ‘`.eb`’
- Specific filename is expected in some contexts (when resolving dependencies)
  - Should match with values for `name`, `version`, `toolchain`, `versionsuffix`
  - `<name>-<version>-<toolchain><versionsuffix>.eb`
- GitHub repository: https://github.com/easybuilders/easybuild-easyconfigs

Step-wise installation procedure



- EasyBuild framework defines step-wise installation procedure, leaves some unimplemented
- Easyblock completes the implementation, override or extends installation steps where needed
- Easyconfig file provides the details (software version, dependencies, toolchain, ...)

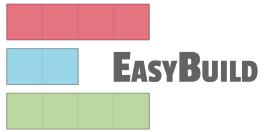
EasyBuild terminology speed run: easystack file

- New concept since EasyBuild v4.3.2 (Dec'20), stable since EasyBuild 5.0
- Concise description for software stack to be installed (in YAML syntax)
- Basically **specifies a set of easyconfig files**
- Specific EasyBuild configuration options can be used per easyconfig file

```
- example.eb:
    options:
        from-commit: d3adb33f  # use easyconfig from specific commit
```

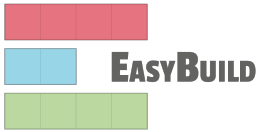
- More info: docs.easybuild.io/easystack-files

EasyBuild terminology speed run: extensions



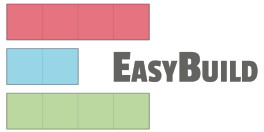
- **Additional software that can be installed *on top* of other software**
- Common examples: Python packages, Perl modules, R libraries, …
- *Extensions* is the general term we use for this type of software packages
- Can be installed in different ways:
  - As a stand-alone software packages (separate module)
  - In a bundle together with other extensions
  - As an actual extension, to provide a “batteries included” installation

EasyBuild terminology speed run: dependencies



- Software that is **required to build/install or run other software**
- **Build dependencies**: only required when building/installing software (not to use it)
  - Examples: CMake, pip, pkg-config, …
- **Dependencies**: (also) required to use the installed software
  - Examples: Python, Perl, R, OpenBLAS, FFTW, …

EasyBuild terminology speed run: toolchains

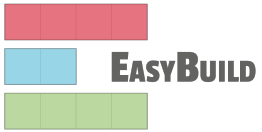


- **Compiler toolchain: set of compilers + libraries for MPI, BLAS/LAPACK, FFT, …**
- Toolchain component: a part of a toolchain (compiler component, etc.)
- **Full toolchain**: C/C++/Fortran compilers + libraries for MPI, BLAS/LAPACK, FFT
- **Subtoolchain** (partial toolchain): compiler-only, only compiler + MPI, etc.
- **System toolchain**: use compilers (+ libraries) provided by the operating system
- **Common toolchains**: widely used toolchains in EasyBuild community:
  - `foss`: GCC + OpenMPI + (FlexiBLAS +) OpenBLAS + FFTW
  - `intel`: Intel compilers + Intel MPI + Intel MKL

EasyBuild terminology speed run: toolchains

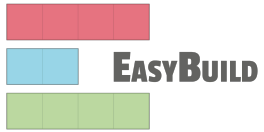
- **LUMI currently does not use the common toolchains**
  - Foss: Still not fully finished with Open MPI, and this may also be holding EESSI back
  - Intel: No upstream support while various versions break in different ways on AMD systems.
- So we have our own toolchains based on the Cray PrgEnv's: cpeGNU, cpeCray, cpeAOCC, cpeAMD.

EasyBuild terminology speed run: modules



- Very overloaded term: kernel modules, Python modules, Perl modules …
- In EasyBuild context: "*module*" usually refers to an **environment module file**
  - **Shell-agnostic specification of how to "activate" a software installation**
  - Expressed in Tcl or Lua syntax (scripting languages)
  - Consumed by a modules tool (**Lmod**, Environment Modules, …)
- Other types of modules will be qualified explicitly (Python modules, etc.)
- EasyBuild automatically generates a module file for each installation

Bringing all EasyBuild terminology together



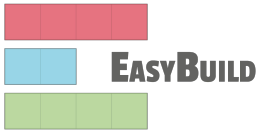
The EasyBuild **framework** leverages **easyblocks** to automatically build and install (scientific) software, potentially including additional **extensions**, using a particular compiler **toolchain**, as specified in **easyconfig files** which each define a set of **easyconfig parameters**.

EasyBuild ensures that the specified **(build) dependencies** are in place, and automatically generates a set of (environment) **modules** that facilitate access to the installed software.

An **easystack** file can be used to specify a collection of software to install with EasyBuild.

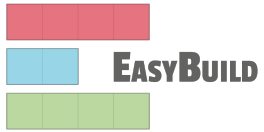
Installing EasyBuild

Installing EasyBuild: requirements



- **Linux** as operating system (CentOS, RHEL, Ubuntu, Debian, SLES, ...)
  - EasyBuild also works on macOS, but support is very basic
- **Python** 3.6+ (Python 3.9+ recommended)
  - Only Python standard library is required for core functionality of EasyBuild
- An **environment modules tool** (`module` command)
  - Default is Lua-based Lmod implementation, highly recommended!
  - Tcl-based implementation (Environment Modules) is also supported

Installing EasyBuild: different options



- Installing EasyBuild using a standard Python installation tool
  - `pip install easybuild`
  - … or a variant thereof (`pip3 install --user`, using `virtualenv`, etc.)
  - May require additional commands, for example to update environment
- **Installing EasyBuild as a module, with EasyBuild *(recommended!)***
  - 2-step “bootstrap” procedure, via temporary EasyBuild installation using `pip`
- Development setup
  - Clone GitHub repositories:

    `easybuilders/easybuild-{framework,easyblocks,easyconfigs}`
  - Update `$PATH` and `$PYTHONPATH` environment variables

Installing EasyBuild: pip install in Python venv

```
eb-demo $ python3 -m venv eb-env
eb-demo $ source eb-env/bin/activate
(eb-env) eb-demo $ pip install --upgrade pip
…
Successfully installed pip-25.1.1
(eb-env) eb-demo $ pip install easybuild archspec rich
Collecting easybuild
…
Installing collected packages: easybuild-framework, easybuild-easyconfigs,
easybuild-easyblocks, easybuild, archspec, rich, …
Successfully installed archspec-0.2.5 easybuild-5.0.0
easybuild-easyblocks-5.0.0 easybuild-easyconfigs-5.0.0
easybuild-framework-5.0.0 rich-14.0.0 …
…
(eb-env) eb-demo $ eb --version
This is EasyBuild 5.0.0 (framework: 5.0.0, easyblocks: 5.0.0) on host
ip-172-31-13-29.eu-central-1.compute.internal.
```

Verifying the EasyBuild installation

- Check EasyBuild version:

```
eb --version
```

- Show help output (incl. long list of supported configuration settings)

```
eb --help
```

- Show the current (default) EasyBuild configuration:

```
eb --show-config
```

- Show system information:

```
eb --show-system-info
```

Updating EasyBuild (with pip or EasyBuild)

- Updating EasyBuild (in-place) that was installed with pip:

```
pip install --upgrade easybuild
```

  (+ additional options like `--user`, or using `pip3`, depending on your setup)

- Use current EasyBuild to install latest EasyBuild release as a module:

```
eb --install-latest-eb-release
```

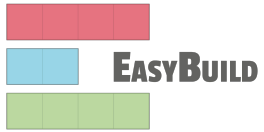
  (you may need to install wheel first: `pip install wheel`)

  - This is *not* an in-place update, but a new EasyBuild installation!
  - You need to load (or swap to) the corresponding module afterwards:

```
module load EasyBuild/5.0.0
```

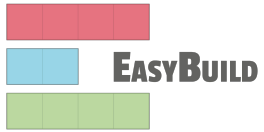
Configuring EasyBuild

Configuring EasyBuild



- EasyBuild should work fine out-of-the-box if you are using Lmod as modules tool
- ... but it will (ab)use `$HOME/.local/easybuild` to install software into, etc.
- **It is *strongly* recommended to configure EasyBuild properly!**
- Main questions you should ask yourself:
  - Where should EasyBuild install software (incl. module files)?
  - Where should auto-downloaded sources be stored?
  - Which filesystem is best suited for software build directories (I/O-intensive)?

Primary configuration settings



- Most important configuration settings: (strongly recommended to specify the ones in **bold**!)
  - Modules tool + syntax (`modules-tool` + `module-syntax`)
  - **Software + modules installation path** (`installpath`)*
  - **Location of software sources "cache"** (`sourcepath`)*
  - **Parent directory for software build directories** (`buildpath`)*
  - Location of easyconfig files archive (`repositorypath`)*
  - Search path for easyconfig files (`robot-paths` + `robot`)
  - Module naming scheme (`module-naming-scheme`)
- Several locations* (+ others) can be controlled at once via `prefix` configuration setting
- *Full* list of EasyBuild configuration settings (~270) is available via `eb --help`

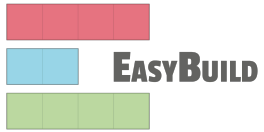
Configuration levels

- There are 3 different configuration levels in EasyBuild:
  - **Configuration files**
  - **Environment variables**
  - **Command line options to the `eb` command**

- Each configuration setting can be specified via each “level” (no exceptions!)

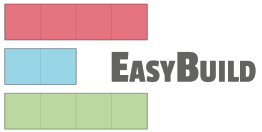
- Hierarchical configuration:
  - Configuration files override default settings
  - Environment variables override configuration files
  - eb command line options override environment variables

EasyBuild configuration files



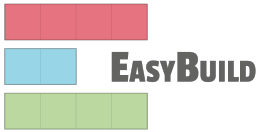
- EasyBuild configuration files are in standard INI format (`key=value`)
- EasyBuild considers multiple locations for configuration files:
  - User-level: `$HOME/.config/easybuild/config.cfg` (or via `$XDG_CONFIG_HOME`)
  - System-level: `/etc/xdg/easybuild.d/*.cfg` (or via `$XDG_CONFIG_DIRS`)
  - See output of `eb --show-default-configfiles`
- Output produced by `eb --confighelp` is a good starting point
- Typically for “do once and forget” static configuration (like modules tool to use, ...)
- **EasyBuild configuration files and easyconfig files are very different things!**

$EASYBUILD_* environment variables



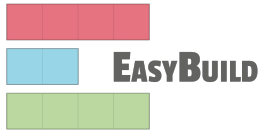
- Very convenient way to configure EasyBuild
- **There is an `$EASYBUILD_*` environment variable for each configuration setting**
  - Use all capital letters
  - Replace every dash (–) character with an underscore (_)
  - Prefix with `EASYBUILD_`
  - Example: `module-syntax` → `$EASYBUILD_MODULE_SYNTAX`
- Common approach: using a shell script or module file to (dynamically) configure EasyBuild

Command line options for `eb` command



- **Configuration settings specified as command line option always "win"**
- Use double-dash + name of configuration setting, like `--module-syntax`
- Some options have a corresponding shorthand (`eb --robot` == `eb -r`)
- In some cases, only command line option really makes sense (like `eb --version`)
- Typically used to control configuration settings for current EasyBuild session; for example: `eb --installpath /tmp/$USER`

Inspecting the current configuration



- It can be difficult to remember how EasyBuild was configured
- Output produced by **`eb --show-config`** is useful to remind you
- Shows configuration settings that are different from default
- Always shows a couple of key configuration settings
- Also shows on which level each configuration setting was specified
- Full current configuration: `eb --show-full-config`

Inspecting the current configuration: example

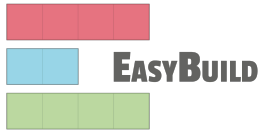
```
$ cat $HOME/config.cfg
[config]
prefix=$HOME/easybuild
buildpath=/tmp/$USER

$ export EASYBUILD_CONFIGFILES=$HOME/config.cfg

$ eb --installpath=/tmp/$USER --show-config
# Current EasyBuild configuration
# (C: command line argument, D: default value,
#  E: environment variable, F: configuration file)
buildpath       (F) = /tmp/ec2-user
configfiles     (E) = /home/ec2-user/config.cfg
containerpath   (F) = /home/ec2-user/easybuild/containers
installpath     (C) = /tmp/ec2-user
packagepath     (F) = /home/ec2-user/easybuild/packages
prefix          (F) = /home/ec2-user/easybuild
repositorypath  (F) = /home/ec2-user/easybuild/ebfiles_repo
robot-paths     (D) = /home/ec2-user/eb-env/easybuild/easyconfigs
rpath           (D) = True
sourcepath      (F) = /home/ec2-user/easybuild/sources
```

Basic usage of EasyBuild

Loading EasyBuild on LUMI

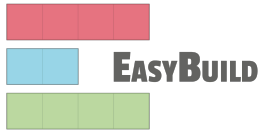
- Set where you want your user software installation through EBU_USER_PREFIX
- Load the LUMI and partition module you want to install software for
- Load the EasyBuild-user module
  - Never load the EasyBuild module directly or you will get the default configuration that does not build upon the LUMI stacks
  - Configures a lot through environment variables

Basic usage of EasyBuild



- **Use `eb` command to run EasyBuild**
- Software to install is usually specified via name(s) of easyconfig file(s), or easystack file
- `--robot` (`-r`) option is required to also install missing dependencies (and toolchain)
- Typical workflow:
  - Find or create easyconfig files to install desired software
  - Inspect easyconfigs, check missing dependencies + planned installation procedure
  - Double check current EasyBuild configuration
  - Instruct EasyBuild to install software (while you enjoy a coffee… or two)

Specifying easyconfigs to use



- The different ways to specify to the `eb` command which easyconfigs to use
  - Specific relative/absolute paths to (directory with) easyconfig files
  - Names of easyconfig files (triggers EasyBuild to search for them)
  - Easystack file to specify a whole stack of software to install (via `eb --easystack`)
- Easyconfig filenames only matter when missing dependencies need to be installed
  - "Robot" mechanism searches based on dependency specs + easyconfig filename
- `eb --search` can be used to quickly search through available easyconfig files:

```
$ eb --search BCFtools
```

Inspecting easyconfigs via `eb --show-ec`

- To see the contents of an easyconfig file, you can use `eb --show-ec`
- No need to know where it is located, EasyBuild will do that for you!

```
$ eb --show-ec BCFtools-1.18-GCC-12.3.0.eb
easyblock = 'ConfigureMake'

name = 'BCFtools'
version = '1.18'

homepage = 'https://www.htslib.org/'
description = """Samtools is a suite of programs for interacting with high-throughput
sequencing data.
 BCFtools - Reading/writing BCF2/VCF/gVCF files and calling/filtering/summarising SNP and
short indel sequence
 variants"""

toolchain = {'name': 'GCC', 'version': '12.3.0'}
toolchainopts = {'pic': True}
…
```

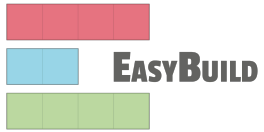
Checking dependencies via `eb --dry-run`

To check which dependencies are required, you can use `eb --dry-run --robot` (or `eb -D -r` or `eb -Dr`):

- Provides overview of all dependencies (both installed and missing)
- Including compiler toolchain and build dependencies

```
$ eb BCFtools-1.18-GCC-12.3.0.eb -Dr
 ...
 * [x] $CFGS/x/XZ/XZ-5.4.2-GCCcore-12.3.0.eb (module: XZ/5.4.2-GCCcore-12.3.0)
 * [x] $CFGS/g/GSL/GSL-2.7-GCC-12.3.0.eb (module: GSL/2.7-GCC-12.3.0)
 * [x] $CFGS/h/HTSlib/HTSlib-1.18-GCC-12.3.0.eb (module: HTSlib/1.18-GCC-12.3.0)
 * [ ] $CFGS/b/BCFtools/BCFtools-1.18-GCC-12.3.0.eb (module:
BCFtools/1.18-GCC-12.3.0)
```

Checking *missing* dependencies via `eb --missing`



To check which dependencies are still *missing*, use `eb --missing` (or `eb -M`):

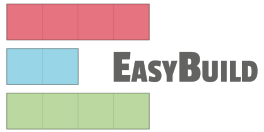
- Takes into account available modules, only shows what is still missing

```
$ eb BCFtools-1.18-GCC-12.3.0.eb -M

1 out of 23 required modules missing:

* BCFtools/1.18-GCC-12.3.0 (BCFtools-1.18-GCC-12.3.0.eb)
```

Inspecting software install procedures

- EasyBuild can quickly unveil how exactly it *would* install an easyconfig file
- Via `eb --extended-dry-run` (or `eb -x`)
- Produces detailed output in a matter of seconds
- Software is not actually installed, all shell commands and file operations are skipped!
- Some guesses and assumptions are made, so it may not be 100% accurate…
- Any errors produced by the easyblock are reported as being ignored
- Very useful to evaluate changes to an easyconfig file or easyblock!

Inspecting software install procedures: example




```
$ eb Boost-1.82.0-GCC-12.3.0.eb -x
...

preparing... [DRY RUN]

[prepare_step method]
Defining build environment, based on toolchain (options) and specified dependencies...

Loading toolchain module...

module load GCCcore/13.2.0 [SIMULATED]
module load binutils/2.40-GCCcore-13.2.0 [SIMULATED]
module load GCC/13.2.0 [SIMULATED]

Loading modules for dependencies...
…
```

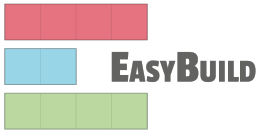
Inspecting software install procedures: example

```
$ eb Boost-1.82.0-GCC-12.3.0.eb -x
...
Defining build environment...
  ...
  export CXX='g++'
  export CXXFLAGS='-O2 -ftree-vectorize -march=native -fno-math-errno -fPIC'
  ...

configuring... [DRY RUN]

[configure_step method]
  running shell command "./bootstrap.sh --with-toolset=gcc
--prefix=/home/user/software/Boost/1.82.0-GCC-12.3.0 --without-libraries=python,mpi"
  (in /tmp/cvanleeuwe/build/Boost/1.82.0/GCC-12.3.0)
  ...
```

Inspecting software install procedures: example

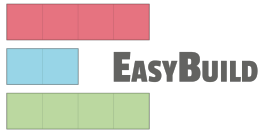
```
$ eb Boost-1.82.0-GCC-12.3.0.eb -x
...

[sanity_check_step method]
Sanity check paths - file ['files']
  * lib/libboost_system-mt-x64.so
  * lib/libboost_system.so
  * lib/libboost_thread-mt-x64.so
Sanity check paths - (non-empty) directory ['dirs']
  * include/boost
Sanity check commands
  (none)

...
```

Installing software with EasyBuild



- To install software with EasyBuild, just run the `eb` command:
  - `eb BCFtools-1.18-GCC-12.3.0.eb`
- If any dependencies are still missing, you will need to also use `--robot`:
  - `eb SAMtools-1.18-GCC-12.3.0.eb --robot`
- More details while the installation is running via trace output (default in EasyBuild v5.x)
  - `eb BCFtools-1.18-GCC-12.3.0.eb --robot --trace`
- To reinstall software, use `eb --rebuild` (or `eb --force`)

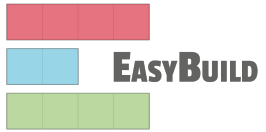
Using software installed with EasyBuild



To use the software you installed with EasyBuild, load the corresponding module:

```
# inform modules tool about modules installed with EasyBuild
module use $HOME/easybuild/modules/all
# check for available modules for BCFtools
module avail BCFtools
# load BCFtools module to "activate" the installation
module load BCFtools/1.18-GCC-12.3.0
```

Stacking software installations

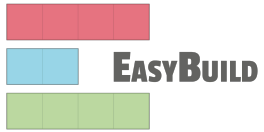


- It's easy to "stack" software installed in different locations
- EasyBuild doesn't care much where software is installed
- As long as the required modules are available to load, it can pick them up
- End users can easily manage a software stack on top of what's installed centrally!

```
module use $HOME/easybuild/modules/all

eb --installpath $HOME/easybuild my-software.eb
```

EasyBuild 5

EasyBuild v5.0

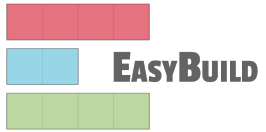


- **Released on 18 March 2025**
- Concludes a development effort that was started in March 2023 (103 weeks)
- Development done in separate `5.0.x` branches, kept in sync with `develop`
- 1,364 merged pull requests

  (framework: 245, easyblocks: 345, easyconfigs: 804)
- **There will be no more EasyBuild 4.x releases,**
  so you must migrate to EasyBuild v5.x!

https://docs.easybuild.io/easybuild-v5

EasyBuild v5.0: Breaking changes

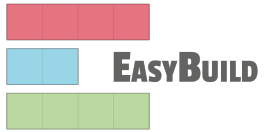


- **Python 3.6+ is required** to run EasyBuild v5.0.0
  - Python 2.7 no longer supported to *run* EasyBuild with (EOL since 2020)

- Updated version requirement for modules tool being used:
  - For Lmod version >= 8.0 is required
  - For Environment Modules version >= 4.3.0 is required

https://docs.easybuild.io/easybuild-v5

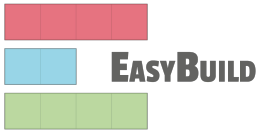
EasyBuild v5.0: Changed defaults



- **RPATH linking** is enabled by default
- Trace output is enabled by default
- `extensions` statement is included by default in generated modules
- `depends_on` is used by default for dependencies in generated modules
- Slurm is used as default job backend
- Default maximum build parallelism is set to 16
- `use_pip` + `sanity_pip_check` enabled by default for `PythonPackage` easyblock
- `CMakeMake` easyblock sets `LIBDIR` configuration option to `lib` by default

https://docs.easybuild.io/easybuild-v5

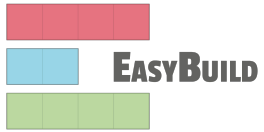
EasyBuild v5.0: Changed behaviour *(selected)*



- `--robot` (`-r`) is no longer enabled by default when using `--dry-run` (`-D`) => Use `eb -Dr`
- Verifying of checksums was moved from from `source` to `fetch` step, to include it with `--fetch`
- `lib` to `lib64` symlink (and vice versa) created before running `postinstallcmds`
- Parsing order for files in `$XDG_CONFIG_DIRS` is reversed + default value is fixed (`/etc/xdg`)
- Unresolved templates in easyconfig parameters are not allowed by default
- Don't automatically prepend a dash (`-`) to first compiler option (relevant for `optarch`)
- Run sanity checks commands from an empty tmpdir rather than the software install directory
- Only allow use of `rpath` toolchain option when `system` toolchain is used

https://docs.easybuild.io/easybuild-v5

EasyBuild v5.0: Enhancements (1/2)

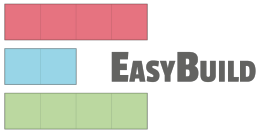


- **New function to run shell commands: `run_shell_cmd`**
- **Interactive debugging of failing shell commands via `env.sh` and `cmd.sh` scripts**
- New collection of easyconfig templates
- Support for installing extensions in parallel stable (no longer experimental)
- Easystack support stable (no longer experimental)
- **Reproducible tarballs for sources created via `git_config` (across Linux & macOS!)**
- New home for the archive of easyconfigs: easybuilders/easybuild-easyconfigs-archive
- Granular exit codes (exit 22 when sanity check fails, exit 31 for missing dependency, ...)
- Copy build directory and/or log file(s) if installation failed to path specified

  via `--failed-install-build-dirs-path` or `--failed-install-logs-path`
- **Specify changes that should be made by generated module files via `module_load_environment`**

https://docs.easybuild.io/easybuild-v5

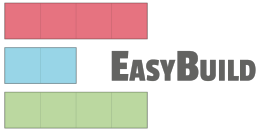
EasyBuild v5.0: Enhancements (2/2)



- Add support for alternate easyconfig parameters/templates/constants
- `keep-debug-symbols` configuration option to set default value of `debug` toolchain option
- Provide control over how generated modules update search path for header files (`$CPATH` or not)
- Provide control over how EasyBuild specifies path to header files during installation
- Provide control over how EasyBuild specifies path to libraries during installation
- Support not using `$PYTHONPATH` to specify the location of installed Python packages
- Revamp of easyconfig parameter `modextrapaths`
- Detect Fortran `.mod` files in installations using `GCCcore` toolchain
- **Let `ConfigureMake` generic easyblock error out on unrecognized configure options**
- Require `download_instructions` for non-public sources

https://docs.easybuild.io/easybuild-v5

EasyBuild v5.0: Removed functionality

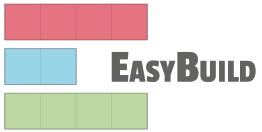


Features that were deprecated in EasyBuild 4.x have been removed:

- EasyBuild bootstrap script
- Experimental support for .yeb easyconfig
- Configuration settings: `accept-eula`, `wait-on-lock` (replaced by equivalent settings)
- Removed functions: `is_generic_easyblock`, `copytree`, `rmtree2`
- Removed methods: `EasyBlock.fetch_extension_sources`, `Toolchain.add_dependencies`
- `mod_exists_regex_template` option in `ModulesTool.exist` method
- Removed options for various methods and functions, like `use_git_am` option for `apply_patch`
- `dummy` toolchain (replaced with `system` toolchain)
- Support for 32-bit targets

https://docs.easybuild.io/easybuild-v5

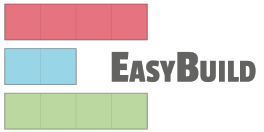
EasyBuild v5.0: Deprecated functionality (1/2)



- **`parallel` easyconfig parameter**
- **`run_cmd` and `run_cmd_qa` functions (replaced with `run_shell_cmd`)**
- `source` step (renamed to `extract`)
- `post_install_step` method in EasyBlock class (renamed to `post_processing_step`)
- Various methods in EasyBlock class: `make_module_req_guess`, `run`, `prerun`, `postrun`, `run_async`
- `easybuild.tools.py2vs3` module (no longer useful since Python 2 is no longer supported)
- Older checksum types

https://docs.easybuild.io/easybuild-v5

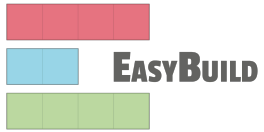
EasyBuild v5.0: Deprecated functionality (2/2)



- `EnvironmentModulesC` or `EnvironmentModulesTcl` modules tools
- GC3Pie as job backend
- Using `optarch` value without leading dash
- `COMPILER*_FLAGS` attributes in `Compiler` class
- Easyconfig parameters: `modextrapaths_append`, `allow_append_abs_path`, `allow_prepend_abs_path`

https://docs.easybuild.io/easybuild-v5

EasyBuild v6.0



- **ETA March 2027** (~2 years after last major EasyBuild release)
- Expected changes (not set in stone yet):
  - Python 3.9+ required (+ recent version of Lmod/Environment Modules)
  - Improved consistency in naming of easyconfig parameters,

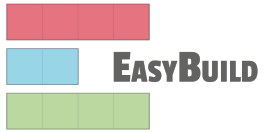
    EasyBuild configuration options, etc.
    - Already partially supported in EasyBuild 5.0.0,

      for example: `configure_opts` instead of `configopts`

PART II: Using EasyBuild

Troubleshooting

Troubleshooting failing installations



- Sometimes stuff still goes wrong…
- Being able to troubleshoot a failing installation is a useful/necessary skill
- Problems that occur include (but are not limited to):
  - Missing source files
  - Missing dependencies (perhaps overlooked required dependencies)
  - Failing shell commands (non-zero exit status)
  - Running out of memory or storage space
  - Compiler errors (or crashes)
- EasyBuild keeps a thorough log for each installation which is very helpful

Troubleshooting: error messages

- When EasyBuild detects that something went wrong, it produces an error
- Very often due to a shell command that produced a non-zero exit code…
- Sometimes the problem is clear directly from the error message:

```
== building…
...
== FAILED: Installation ended unsuccessfully: shell command 'make ...' failed
with exit code 2 in build step for BCFtools-troubleshooting.eb (took 3 secs)
```

- It may take a bit of effort to figure out the *actual* underlying problem

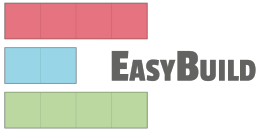
Troubleshooting: log files

- EasyBuild keeps track of the installation in a detailed log file
- During the installation, it is stored in a temporary directory:

```
$ eb example.eb
== Temporary log file in case of crash /tmp/eb-r503td0j/easybuild-17flov9v.log
...
```

- Includes executed shell commands and output, build environment, etc.
- More detailed log file when debug mode is enabled (`debug` configuration setting)
- There is a log file per EasyBuild session, and one per performed installation
- When an installation completes successfully, the log file is copied to a subdirectory of the software installation directory
- **Trick** when failure: vim $(eb –last-log)

Troubleshooting: navigating log files



- **EasyBuild log files are well structured, and fairly easy to search through**
- Example log message, showing prefix ("== "), timestamp, source location, log level:

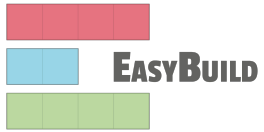
```
== 2025-05-19 08:43:21,688 run.py:500 INFO Running shell command 'make  -j 16

CFLAGS="-O2 -faster"' in /tmp/ec2-user/BCFtools/1.18/GCC-12.3.0/bcftools-1.18
```

- Different steps of installation procedure are clearly marked:

```
== 2025-05-19 08:43:21,817 example INFO Starting sanity check step
```

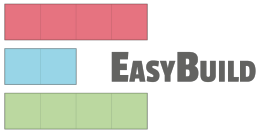
- To find actual problem for a failing shell command, look for patterns like:
  - ERROR
  - Error 1
  - error:
  - failure
  - not found
  - No such file or directory
  - Segmentation fault

Troubleshooting: inspecting the build directory



- EasyBuild leaves the build directory in place when the installation failed
- Can be useful to inspect the contents of the build directory for debugging
- For example:
  - Check `config.log` when `configure` command failed
  - Check `CMakeFiles/CMakeError.log` when `cmake` command failed (good luck...)

Troubleshooting with EasyBuild v5.0



- **EasyBuild v5.0 makes troubleshooting failing installations significantly easier**
- When a shell command run by EasyBuild fails:
  - The problem will be reported in a more user-friendly way
  - You can quickly inspect (only) the output of that command
  - A script is generated to start an **interactive shell session** to debug “in context”: in the correct working directory + prepared build environment
- Made possible by switching to new `run_shell_cmd` function

Improved error reporting in EasyBuild v5.0

EasyBuild 5.0 produces clearer error messages when a shell command failed:

```
ERROR: Shell command failed!
       full command                -> make  -j 8 LDFLAGS='-lfast'
       exit code                   -> 2
       called from                 -> 'build_step' function in /.../easyblocks/generic/configuremake.py (line 357)
       working directory           -> /tmp/ec2-user/kenneth/easybuild/build/BCFtools/1.18/GCC-12.3.0/bcftools-1.18
       output (stdout + stderr)  -> /tmp/eb-i61vle8x/run-shell-cmd-output/make-1ynysa6f/out.txt
       interactive shell script  -> /tmp/eb-i61vle8x/run-shell-cmd-output/make-1ynysa6f/cmd.sh
```

- Colors to draw attention to the most important parts of the error message
- File with (only) command output + path to build directory are easy to find
- **Auto-generated `cmd.sh` script starts interactive subshell in correct build environment!**

This is powered by the new `run_shell_cmd` function that EasyBuild uses to run shell commands, which took a lot of effort, partially because all ~240 easyblocks has to be updated to use `run_shell_cmd`.

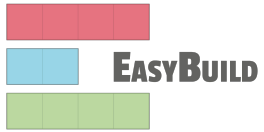
Example: What could be going on here?

```
$ eb example.eb
…
== building…
== FAILED: Installation ended unsuccessfully (build directory:
/tmp/example/example/1.0/GCC-10.2.0):
build failed (first 300 chars): cmd "make" exited with exit code 2 and output:
/usr/bin/g++ -O2 -ftree-vectorize -march=znver2 -fno-math-errno -c -o core.o core.cpp
cc1plus: error: bad value (‘znver2’) for ‘-march=’ switch (took 1 sec)
== Results of the build can be found in the log file(s)
/tmp/eb-dbobppfh/easybuild-example-1.0-20200613.145414.aUEJA.log
ERROR: Build of /home/easybuild/subread.eb failed (err: ...)
```

- Here it is using the system gcc…
  Happens when software does not respect the usual environment variables to select the compiler or has the compiler hard-coded in the configure scripts.

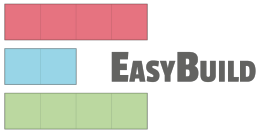
Creating EasyConfig files

Adding support for additional software



- Every installation performed by EasyBuild requires an easyconfig file
- Easyconfig files can be:
  - Included with EasyBuild itself (or obtained elsewhere)
  - Derived from an existing easyconfig (manually or automatic)
  - Created from scratch
- Most easyconfigs leverage a generic easyblock
- Sometimes using a custom software-specific easyblock makes sense...

Easyblocks vs easyconfigs



- When can you get away with using an easyconfig leveraging a generic easyblock?
- When is a software-specific easyblock really required?
- Easyblocks are *"implement once and forget"* (at least in theory, maintenance can be a pain...)
- Easyconfig files leveraging a generic easyblock can become too complicated (subjective)
- Reasons to consider implementing a custom easyblock:
  - 'critical' values for easyconfig parameters required to make installation succeed (e.g., bowtie2)
  - Toolchain-specific options (e.g., CP2K: Extra flags for the GNU compiler)
  - custom (configure) options related to toolchain or included dependencies (e.g., VMD)
  - interactive commands that need to be run (e.g., Maple)
  - having to create or adjust specific (configuration) files (e.g., Siesta)
  - 'hackish' usage of a generic easyblock
  - complex or very non-standard installation procedure (e.g., GCCcore in EasyBuild)

Writing easyconfig files

- Collection of easyconfig parameter definitions (Python syntax), collectively specify what to install
- Some easyconfig parameters are **mandatory**, and must always be defined:
  `name`, `version`, `homepage`, `description`, `toolchain`
- Commonly used easyconfig parameters (but strictly speaking not required):
  - `easyblock` (by default derived from software name)
  - `versionsuffix`
  - `source_urls`, `sources`, `patches`, `checksums`
  - `dependencies`, `builddependencies`
  - `preconfigopts`, `configopts`, `prebuildopts`, `buildopts`, `preinstallopts`, `installopts`
  - `sanity_check_paths`, `sanity_check_commands`

Mandatory parameters (and some closely related ones)

```
name = 'example'
version = '1.0'
versionsuffix = '-OpenMP'

homepage = 'https://example.org'

description = """
This is an example
of a multi-line description.
It is spread across multiple lines.
"""

toolchain = SYSTEM
toolchain = {'name': 'cpeGNU', 'version': '24.03'}
```

On LUMI:

```
whatis = ['Description: Blosc is an extremely fast, multi-threaded, meta-compressor library']
```

Generic easyblocks

- Need to be specified with the easyblock parameter
- ConfigureMake: Standard ./configure, make, make install procedure
- CMakeMake: Uses CMake for configuring, but make and make install for build and installation
- PythonPackage: A single Python package
- Bundle: Can bundle a set of packages in a single installation. Each package can use its own generic or custom easyblock
- PythonBundle
- But there are many more...

EasyBlocks and extra parameters

- `eb --list-easyblocks`
- `eb -a -e ConfigureMake`
- Popular ones in many easyconfigs:
  - `preconfigopts` and `configopts`
    `<preconfigopts> ./configure <options from EasyBuild> <configopts>`
  - `prebuildopts` and `buildopts`
  - `preinstallopts` and `installopts`
  - Tip: use `&&` to glue commands together so that the command as a whole fails if one of the subcommands fails.

    ```
    preconfigopts = 'HDF5_PREFIX="$EBROOTHDF5" &&'
    ```

Toolchain options

```
toolchainopts = {'usempi': False, 'openmp': False, 'extra_cxxflags': '-std=c++11'}
```

Many options, see also the technical documentation

- `usempi`: Currently doesn't really matter on LUMI
- `openmp`: Add the compiler flag for OpenMP to CFLAGS etc.
- `pic`: Enable position-independent code
- `cstd`: Set the C and C++ standard flags in CFLAGS and CXXFLAGS
- `verbose`: Verbose output of the compiler
- `extra_cflags`, `extra_cxxflags`, `extra_f90flags`, `extra_fcflags`, `extra_fflags`

Specifying sources

```
source_urls = [
    'https://example.org/download/',
    'https://example.org/download/archive/',
]
sources = ['example-1.0-src.tar.gz']
patches = ['example-fix.patch']
checksums = [
    '9febae18533d035ac688d977cb2ca050e6ca8379311d7a14490ad1ef948d45fa',
    '864395d648ad9a5b75d1a745c8ef82b78421d571584037560a22a581ed7a261c',
]

sources = ['example-%(version)s-src.tar.gz']
```

Dependencies

- Runtime dependencies: Module needs to be loaded at runtime also: parameter `dependencies`
- Build dependencies: Only needed when building the software: parameter `builddependencies`
- No concept yet for link dependencies
  - They are a build dependency when linking statically or using rpath
  - But a runtime dependency when using shared libraries and `LD_LIBRARY_PATH`
- OS dependencies: Checks if certain packages are installed in the OS.

Dependencies

```
builddependencies = [
  ('buildtools', '%(toolchain_version)s', '', SYSTEM)
]

dependencies = [
    ('cray-hdf5',   EXTERNAL_MODULE),
    ('cray-netcdf', EXTERNAL_MODULE),
    ('GSL',         '2.7.1'),
    ('ANTLR',       '2.7.7', '-cray-python3.11'),
]

osdependencies = [
    ('openssl-devel', 'libssl-dev', 'libopenssl-devel')
]
```

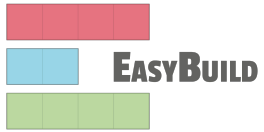
Checking the result

```
sanity_check_paths = {
    'files': ['bin/example'],
    'dirs': ['examples/one', 'examples/two'],
}

sanity_check_commands = [
    "example --version",
    "example --help",
]
```

Modules

- Common parameter imposed by many naming schemes:
  ```
  moduleclass = 'lib'
  ```
- Extra variables
  ```
  modextravars = {'NCCL_SOCKET_IFNAME': 'hsn'}
  ```
- Extra path-style variables:
  ```
  modextrapaths = {'INFOPATH' : 'share/info'}
  ```
- Extra LUA code
  ```
  modluafooter = """
  local start_vnc_bash = 'source %(installdir)s/scripts/VNC/_get_vnc_display "$@"'
  local start_vnc_csh  = 'source %(installdir)s/scripts/VNC/_get_vnc_display $*'
  set_shell_function( 'start-vnc', start_vnc_bash, start_vnc_csh )
  """
  ```

Generating tweaked easyconfig files



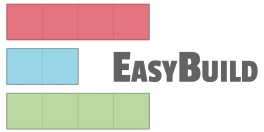
- Trivial changes to existing easyconfig files can be done automatically
- Bumping software version: `eb example-1.0.eb --try-software-version 1.1`
- Changing toolchain (version): `eb example.eb --try-toolchain GCC,12.3.0`
- Changing specific easyconfig parameters (limited): `eb --try-amend ...`
- Note the “try” aspect: additional changes may be required to make installation work
- EasyBuild does save the so generated easyconfig files in the `easybuild` subdirectory of the software installation directory and in the easyconfig archive.

Copying easyconfig files

- Small but useful feature: copy specified easyconfig file via `eb --copy-ec`
- Avoids the need to locate the file first via `eb --search`
- Typically used to create a new easyconfig using existing one as starting point
- Example:

```
$ eb --copy-ec BCFtools-1.18-GCC-12.3.0.eb BCFtools.eb
...
BCFtools-1.18-GCC-12.3.0.eb copied to BCFtools.eb
```

Exercise on creating easyconfig file from scratch



- Step-wise example + exercise of creating an easyconfig file from scratch
- For fictitious software packages: `eb-tutorial` + `py-eb-tutorial`
- Sources available at https://github.com/easybuilders/easybuild-tutorial/tree/main/docs/files
- **Great exercise to work through these yourself!**

```
name = 'eb-tutorial'

version = '1.0.1'

homepage = 'https://easybuilders.github.io/easybuild-tutorial'

description = "EasyBuild tutorial example"
```

Using external modules from the Cray PE

External modules

- Concept developed for CSCS Piz Daint: Integrate with a module that is not generated by EasyBuild and add some metadata for that module so that to EasyBuild it acts like a regular EasyBuild module
  - When loaded in EasyBuild, EasyBuild will generate EBROOT and EBVERSION variables for the package as if it is an EasyBuild package
  - E.g., when using cray-fftw it will define EBROOTFFTW and EBVERSIONFFTW so that to the installation process it looks like the EasyBuild FFTW module
- Set via `EASYBUILD_EXTERNAL_MODULES_METADATA` in EasyBuild-user etc.
- Example:

```
[cray-fftw]
name = FFTW
prefix = FFTW_DIR/..
version = 3.3.8.10
```

Use as a dependency

- Usually without a version:

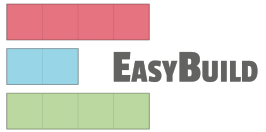
```
dependencies = [('cray-fftw', EXTERNAL_MODULE)]
```

- But specifying a version is possible:

```
dependencies = [('cray-fftw/3.3.8.12', EXTERNAL_MODULE)]
```

Implementing EasyBlocks

Implementing Easyblocks



- An easyblock may be required for more complex software installations
- This requires some Python skills, and familiarity with EasyBuild framework
- A software-specific easyblock can derived from a generic easyblock
- Focus is usually on configure/build/installs steps of installation procedure
- See the notes
- See also https://docs.easybuild.io/implementing-easyblocks

Custom EasyBlocks on LUMI

- Some standard EasyBlocks don't work well on LUMI because they test on compiler and fail if they don't know the compiler
  - We rename them when we adapt them for our purposes
- We do have some of our own, but solve many issues in the EasyConfig instead
- Hard to maintain: They have to cover multiple versions of software and multiple versions of EasyBuild
  - Will likely have to break the structure of our repository and start an EasyBlock repository for specific versions of the LUMI software stacks (or of EasyBuild)
  - Which will break GitHub integration
  - Our current rocm EasyBlock breaks some of the oldest versions of EasyBuild on LUMI...

Part III: Advanced topic

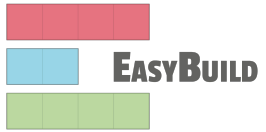
EasyBuild as a library

EasyBuild as a library

- It is possible to use EasyBuild as a library also, directly calling its Python functions.
- One use case could be to parse an EasyConfig file without installing the software, but using the information, e.g., to generate documentation pages based on the value of some EasyConfig parameters.
- Not used on LUMI at the moment, but Orian did experiment with it in the early days of LUMI

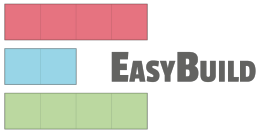
Hooks

Customizing EasyBuild via Hooks



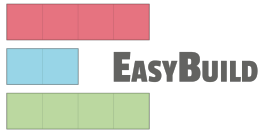
- Hooks allow you to customize EasyBuild easily and consistently
- Set of Python functions that are automatically picked up by EasyBuild
- Can be used to ”hook” custom code into specific installation steps
- Make EasyBuild use your hooks via `hooks` configuration option
- Examples:
  - Inject or tweak configuration options
  - Change toolchain definitions
  - Custom checks to ensure that site policies are taken into account

Customizing EasyBuild via Hooks



- Types
  - Start and end hooks: Run at the start of EasyBuild and at the very end
  - Parse hooks: Triggered after the parsing and the point to inject new code or modify code in the EasyConfigs
  - Pre and post hooks for most installation steps
- Care needed when writing them. You can break EasyBuild and making changes in a datastruture can be different from what you expect
- Extensively documented: docs.easybuild.io/hooks

Hooks: examples



- EUM'22 talk by Alex: Building a heterogeneous MPI stack with EasyBuild

  https://easybuild.io/eum22/#eb-mpi

- `contrib/hooks` subdirectory in easybuild-framework GitHub repository:

  https://github.com/easybuilders/easybuild-framework/tree/develop/contrib/hooks

Hooks: examples

Ensure that software is installed with a specific license group:

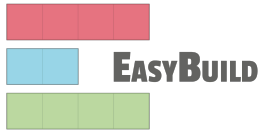
```
def parse_hook(self, *args, **kwargs):
    if self.name == 'example':
        # use correct license group for software ‘example’
        self['group'] = 'licensed_users_example'
```

Hooks on LUMI

- Parse hook: Some code that extends the EasyConfigs for the toolchains themselves with proper target modules
- Parse hook: Detect when a user is trying to use the wrong toolchain
- Parse hook: Add a site_contacts line for all pre-installed software
- End hook: Clear the Lmod user cache

Slurm integration

Submitting Installations as Slurm Jobs



- EasyBuild can *distribute* the installation of a software stack as jobs
- Slurm is the default job backend in EasyBuild v5.x
- Use "`eb ... --job --robot`" to submit software installations to be performed with EasyBuild as Slurm jobs
- Expect some issues on LUMI with the current choice of the temporary directories
- And useless for managing the central software stack on LUMI
- See also https://docs.easybuild.io/submitting-jobs

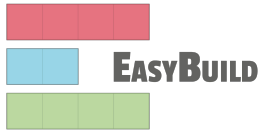
Module naming schemes

Module naming schemes

- See the notes
- We use a slightly adapted naming scheme on LUMI.
  - Flat namespace in each stack
  - Removed the "all" from the path and the copies in a moduleclass. This was a stupid idea at the start of EasyBuild that no site seems to use as many modules belong in more than one class which is not supported by EasyBuild

Github integration

Contributing to EasyBuild



There are several ways to contribute to EasyBuild, including:

- Providing feedback (positive or negative)
- Reporting bugs
- Joining the discussions (mailing list, Slack, conf calls)
- Sharing suggestions/ideas for enhancements & additional features
- Contributing easyconfigs, enhancing easyblocks,

  adding support for new software, implementing additional features, ...
- Extending & enhancing documentation

GitHub integration features

- EasyBuild has strong integration with GitHub, which facilitates contributions
- Some additional Python packages required for this: GitPython, keyring
- Also requires some additional configuration, incl. providing a GitHub token
- **Enables creating, updating, reviewing pull requests using `eb` command!**
- Makes testing contributions very easy: ~2,500 easyconfig pull requests per year!
- Extensively documented:
  docs.easybuild.io/integration-with-github

Opening a pull request in 1, 2, 3

```
$ mv sklearn.eb scikit-learn-1.4.2-gfbf-2023a.eb
$ mv scikit-learn*.eb easybuild/easyconfigs/s/scikit-learn
$ git checkout develop && git pull upstream develop
$ git checkout -b scikit_learn_142_gfbf_2023a
$ git add easybuild/easyconfigs/s/scikit-learn
$ git commit -m "{data}[gfbf/2023a] scikit-learn v1.4.2"
$ git push origin scikit_learn_142_gfbf_2023a
```

+ log into GitHub to actually open the pull request (clickety, clickety...)

one single `eb` command

no git commands

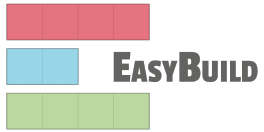
no GitHub interaction

metadata is automatically derived from easyconfig

saves a lot of time!

`eb --new-pr sklearn.eb`

Agenda



- [13:30-13:40] What is EasyBuild?
- [13:40-13:50] Changes in EasyBuild 5.0
- [13:50-14:00] Installation and configuration of EasyBuild (incl. live demo)
- [14:00-14:10] Basic Usage of EasyBuild (incl. live demo)
- [14:10-14:20] Troubleshooting (incl. live demo)
- [14:20-14:35] Adding support for additional software
- [14:35-14:45] Advanced topics: hooks & beyond
- **[14:45-15:00] Using EasyBuild on top of EESSI (incl. live demo)**
- [15:00-15:30] Q&A

Installing something with a lot of dependencies…

- For example, PyTorch…

```
$ eb --missing PyTorch-bundle-2.1.2-foss-2023a.eb
146 out of 164 required modules missing:
* pkgconf/1.9.5-GCCcore-12.3.0 (pkgconf-1.9.5-GCCcore-12.3.0.eb)
* UnZip/6.0-GCCcore-12.3.0 (UnZip-6.0-GCCcore-12.3.0.eb)
* expat/2.5.0-GCCcore-12.3.0 (expat-2.5.0-GCCcore-12.3.0.eb)
...
* sympy/1.12-gfbf-2023a (sympy-1.12-gfbf-2023a.eb)
* PyTorch/2.1.2-foss-2023a (PyTorch-2.1.2-foss-2023a.eb)
* PyTorch-bundle/2.1.2-foss-2023a (PyTorch-bundle-2.1.2-foss-2023a.eb)
```

- That’s going to take forever…

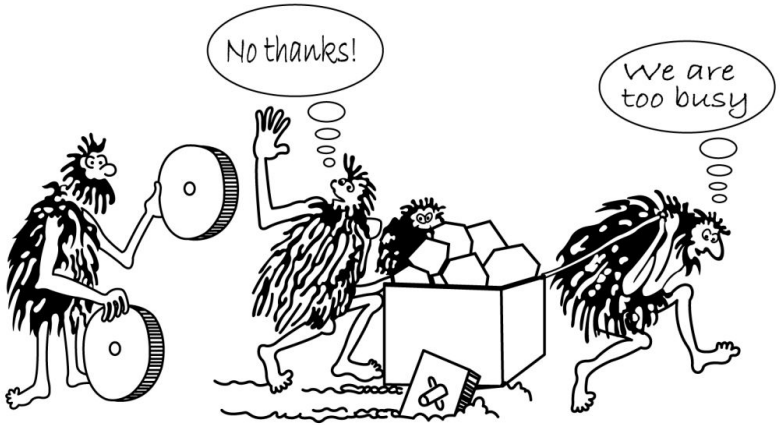
Extra: Building on top of EESSI

What is EESSI?

EASYBUILD

EESSI

EUROPEAN ENVIRONMENT FOR
SCIENTIFIC SOFTWARE INSTALLATIONS



EESSI in a nutshell

- *European Environment for Scientific Software Installations (EESSI)*
- **Shared repository of (optimized!) scientific software *installations***
- Avoid duplicate work across (HPC) sites by collaborating on a shared software stack
- Uniform way of providing software to users, regardless of the system they use!
- Should work on any Linux OS (+ WSL, and macOS*) and system architecture
  - From laptops and personal workstations to HPC clusters and cloud
  - Support for different CPUs, interconnects, GPUs, etc.
- **Focus on performance, automation, testing, collaboration**

https://www.eessi.io/docs/

*through Lima

EESSI ingredients

Compatibility layer

Abstraction from the host OS

Filesystem Layer

Global distribution of software installations

EESSI

EUROPEAN ENVIRONMENT FOR SCIENTIFIC SOFTWARE INSTALLATIONS

Software Layer

Optimized software Installations for specific CPU microarchitectures

Intuitive user interface: module avail, module load, …

Automatic selection of Best suited part of Software stack for CPU microarchitectures

Using EESSI

- Here, we'll assume EESSI is already installed. If not, follow instructions on eessi.io/docs/getting_access/native_installation/
- Set up EESSI environment by loading the module

```
$ ls /cvmfs/software.eessi.io
host_injections  init  README.eessi  versions

$ module unuse $MODULEPATH

$ module use /cvmfs/software.eessi.io/init/modules

$ module load EESSI/2023.06
EESSI/2023.06 loaded successfully...

$ module avail

...
```

EESSI-extend: building on top of EESSI with EasyBuild

- EESSI provides base installations
- We can install on top of the EESSI software stack with EasyBuild


```
$ module load EESSI-extend/2023.06-easybuild

-- Using /tmp/$USER as a temporary working directory for installations, you can override this
by setting the environment variable WORKING_DIR and reloading the module (e.g., /dev/shm is a
common option)

Configuring for use of EESSI_USER_INSTALL under /home/ec2-user/eessi

-- To create installations for EESSI, you _must_ have write permissions to
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4

-- You may wish to configure a sources directory for EasyBuild (for example, via setting the
environment variable EASYBUILD_SOURCEPATH) to allow you to reuse existing sources for
packages.
```

EESSI-extend: building on top of EESSI with EasyBuild

- EESSI provides base installations
- We can install on top of the EESSI software stack with EasyBuild

```
$ eb --missing PyTorch-bundle-2.1.2-foss-2023a.eb
11 out of 138 required modules missing:

* parameterized/0.9.0-GCCcore-12.3.0 (parameterized-0.9.0-GCCcore-12.3.0.eb)
* Scalene/1.5.26-GCCcore-12.3.0 (Scalene-1.5.26-GCCcore-12.3.0.eb)
...
* PyTorch-bundle/2.1.2-foss-2023a (PyTorch-bundle-2.1.2-foss-2023a.eb)
```

- Much more feasible!

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps           (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars       (E) = LD_LIBRARY_PATH
hooks                 (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath           (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                 (D) = True
...
sysroot               (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps            (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars        (E) = LD_LIBRARY_PATH
hooks                  (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath            (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                  (D) = True
...
sysroot                (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

These deps are provided by Gentoo-prefix

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps           (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars       (E) = LD_LIBRARY_PATH
hooks                 (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath           (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                 (D) = True
...
sysroot               (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

Libraries are found at runtime by setting RPATH instead of LD_LIBRARY_PATH (avoids host OS executables from using libs from EESSI)

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps           (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars       (E) = LD_LIBRARY_PATH
hooks                 (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath           (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                 (D) = True
...
sysroot               (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

A set of EasyBuild hooks that are used to fix issues (e.g. software X doesn't build on ARM with vectorization => disable vectorization)

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps           (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars       (E) = LD_LIBRARY_PATH
hooks                 (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath           (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                 (D) = True
...
sysroot               (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

Installpath contains architecture + micro-architecture, as detected by architect. Mimics EESSI directory structure.

EESSI-extend: how does it work?

- Configures EasyBuild very similar to how main EESSI software stack is built

```
$ eb --show-config
...
filter-deps           (E) = Autoconf, Automake, Autotools, binutils, bzip2, DBus, flex,
gettext, gperf, help2man, intltool, libreadline, libtool, M4, makeinfo, ncurses, util-linux,
XZ, zlib
filter-env-vars       (E) = LD_LIBRARY_PATH
hooks                 (E) =
/cvmfs/software.eessi.io/versions/2023.06/init/easybuild/eb_hooks.py
...
installpath           (E) =
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4
...
rpath                 (D) = True
...
sysroot               (E) = /cvmfs/software.eessi.io/versions/2023.06/compat/linux/x86_64
```

Tells EasyBuild that the OS against which it has to build (Gentoo-prefix) resides in this prefix

Environment variables that influence EESSI-extend

WARNING: you need to set the environment variables before loading EESSI-extend

- `$EESSI_CVMFS_INSTALL`
  - to install in the main EESSI prefix (/cvmfs/software.eessi.io/versions/...)
  - for CVMFS admins of the EESSI repository only
  - Umask: 022.
  - Example: `EESSI_CVMFS_INSTALL=1`
- `$EESSI_SITE_INSTALL`
  - install dir will be $EESSI_SITE_SOFTWARE_PATH (default: /cvmfs/software.eessi.io/host_injections/...)
  - for HPC support staff building a local software environment (for end-users) on top of EESSI
  - Umask: 022.
  - Example: `EESSI_SITE_INSTALL=1`

Environment variables that influence EESSI-extend

- `$EESSI_PROJECT_INSTALL`
  - install in `$EESSI_PROJECT_INSTALL/versions/<eessi_version>/software/<os>/$EESSI_SOFTWARE_SUBDIR`
  - for project groups (i.e. end users) to build a common software stack in e.g. a project space
  - Umask: 002. Group-writeable-installdir: true. Set-gid-bit: true. Sticky bit: false.
  - Example: EESSI_PROJECT_INSTALL=/my/project/dir (dir has to exist!)
- `$EESSI_USER_INSTALL` *(default)*
  - install in `$EESSI_USER_INSTALL/versions/<eessi_version>/software/<os>/$EESSI_SOFTWARE_SUBDIR`
  - for individual end-users, to build additional software in their homedir
  - Umask: 077. Sticky bit: true.
  - Example: EESSI_USER_INSTALL=$HOME/my/prefix (dir has to exist!)

EESSI-extend:

- Now, actually install PyTorch-bundle-2.1.2-foss-2023a.eb

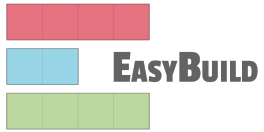
```
$ eb PyTorch-bundle-2.1.2-foss-2023a.eb --robot
...
== COMPLETED: Installation ended successfully (took 14 secs)
== Results of the build can be found in the log file(s)
/home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4/software/PyTorch-bundle/
2.1.2-foss-2023a/easybuild/easybuild-PyTorch-bundle-2.1.2-20241209.133133.log.bz2

== Build succeeded for 11 out of 11
...
$ module av PyTorch-bundle/2.1.2-foss-2023a

------ /home/ec2-user/eessi/versions/2023.06/software/linux/x86_64/amd/zen4/modules/all
-------
   PyTorch-bundle/2.1.2-foss-2023a (D)
...
```

EESSI-extend, final remarks:

- If you don't set any of the `EESSI_*_INSTALL` environment vars, `EESSI-extend` defaults to `EESSI_USER_INSTALL=$HOME/eessi`
- `EESSI-extend` installs and optimizes for the current host
  - Installpath based on current host architecture
  - Default EasyBuild optimization is used: `--optarch=None`, meaning native optimization
  - If your login node has different architecture from your batch nodes, install on a batch node
  - On a heterogenous cluster, you will *probably* want to install everything once per architecture in your cluster
- Modules installed with `EESSI-extend` are only visible *after* loading `EESSI-extend` again

https://www.eessi.io/docs/using_eessi/building_on_eessi/

Questions?



- Website: https://easybuild.io
- Documentation: https://docs.easybuild.io
- Tutorials: https://tutorial.easybuild.io
- **10th EasyBuild User Meeting: https://easybuild.io/eum25 (slides+recording of talks available!)**
- Getting help:
  - Mailing list: https://lists.ugent.be/wws/subscribe/easybuild
  - **Slack: https://easybuild.slack.com - https://easybuild.io/join-slack**
  - Bi-weekly conference calls: https://github.com/easybuilders/easybuild/wiki/Conference-calls

Website: https://eessi.io

Join our Slack channel (see join link on website)

Documentation: https://eessi.io/docs

Blog: https://eessi.io/docs/blog

GitHub: https://github.com/eessi

Paper (open access): https://doi.org/10.1002/spe.3075

EESSI YouTube channel

Bi-monthly online meetings

(first Thu odd months, 2pm CEST)

MultiXscale

Co-funded by the European Union

EuroHPC Joint Undertaking

Web page: multixscale.eu

Facebook: MultiXscale

Twitter: @MultiXscale

LinkedIn: MultiXscale

BlueSky: MultiXscale

KEMIJSKI INŠTITUT

UNIVERSITAT DE BARCELONA

Universität Stuttgart

UNIVERSITAS BERGENSIS

SORBONNE UNIVERSITÉ

Université de Toulouse

Consiglio Nazionale delle Ricerche

MAX-PLANCK-GESELLSCHAFT

HPC Now!

university of groningen

SURF

JÜLICH Forschungszentrum

GHENT UNIVERSITY

BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

LEONARDO

iit ISTITUTO ITALIANO DI TECNOLOGIA

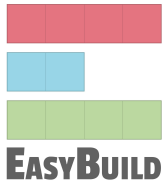
Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and countries participating in the project under grant agreement No 101093169.

Webinar series: Different aspects of EESSI

5 Mondays in a row May-June 2025

https://eessi.io/docs/training/2025/webinar-series-2025Q2

- Introduction to EESSI - **slides+recording available!**
- Introduction to CernVM-FS - **slides+recording available**
- **Introduction to EasyBuild *(today)***
- EESSI for CI/CD *(26 May)*
- Using EESSI as the base for a system stack *(2 June)*



More info and registration →

Part IV: EasyBuild on LUMI

Repositories

- Three LUST-maintained repositories with EasyConfig files
  - LUMI-SoftwareStack: Centrally installed software, but also the whole Lmod configuration and special modules, and scripts to manage the software stack
    - Documented on lumi-supercomputer.github.io/LUMI-SoftwareStack
  - LUMI-EasyBuild-contrib: User-installable software
  - LUMI-EasyBuild-containers: Module wrappers for containers: AMD AI containers and CCPE
- Users can have their own repo also
  - Clone as UserRepo in EBU_USER_PREFIX and it will be used by EasyBuild-user
- LUMI-EasyBuild-docs to manage building the LUMI Software Library

Managing updates

- Work as much as possible with pull requests
  - Hard to work if everybody start to push directly to the main branch…
  - Review possible
  - Easy for me to keep an overview and also update the documentation
- Central software stack is no playground
  - Test in an independent copy
  - Hard to change software once it is in there as it may be in use
  - 5 copies: Master on flash, users load from one of the 4 hard disk based systems
  - Flash only mounted on /appl/lumi on uan06
  - Everybody in the group appl_lumi (462000009) can damage the software stack…
  - If you damage one of the lustrep copies, it will be repaired at the next sync but if damage on lustref remains unnoticed…

Managing updates

- I currently prepare a release on a laptop from a directory with 5 repositories
  - LUMI-SoftwareStack
  - LUMI-EasyBuild-contrib
  - LUMI-EasyBuild-containers
  - LUMI-EasyBuild-docs
  - tags
- And have a script to build a release and push to GitHub
  - Need to rework a bit and share

Documenting the software stack: LUMI Software Library

- Generated by a shell script with some flaws (in particular in dealing with #)
- From files stored with the EasyConfigs of each package:
  - USER.md: Optional, extra information for users
  - LICENSE.md: Highly preferred. Information on the license of the package
  - README.md: Technical information
    - Where on the internet can we find the package?
    - Supported in EasyBuild and/or Spack?
    - What decisions did we take, how did we build the EasyConfig?
- And from lines in the EasyConfig files starting with #DOC
- And some information in LUMI-EasyBuild-docs
  - What's new and Issues section
  - Packages that have no EasyConfig

Documenting the software stack: LUMI Software Library

- Example: QuantumESPRESSO
  - In GitHub
- Example: ELPA: Was moved from centrally installed to contrib
  - Will take some care when the centrally installed version is fully archived
- Example: buildtools
  - Care taken to ensure that users can find all the tools in there via the search box

Documenting the software stack: LUMI Software Library

- I have a script to generate LUMI Software Library-like documentation for a user repo and could add that to LUMI-EasyBuild-contrib and LUMI-EasyBuild-containers also which would testing your contributions to the README.md and other files easier.

Documenting the LUMI software stack: Toolchains etc

- The LUMI-SoftwareStack repository has its own documentation about everything in there except the easyconfigs and easyblocks
  - docs subdirectory
  - docs/config contains the mkdocs.yml file, Makefile and requirements .txt file for the mkdocs installation
  - Automatically updated on github.io when pushing to the main branch, but manual processing possible to check before pushing

Documenting the LUMI Software Stack: Roll-out on LUMI

- Installation done on uan06
- All steps are documented in detail on the "Change log for the software stack in /appl/lumi" page in the LUST eduuni wiki.
  - Done before the actual installation to force me to think already
  - Procedure tested as much as possible in a local copy of the software stack
  - I am VERY strict about this. We're managing a big and expensive infrastructure and each mistake is costly
  - But Orian showed last year that this approach worked...

Managing versions of dependencies

- Consistently updating versions of dependencies is a pain
- It would be nice to get versions from some database or yaml file or so, but EasyBuild does not support that yet
  - They are experimenting with such a thing for checksums though
- Primitive LUST solution:
  - Standard variable names, e.g., local_zlib_version
  - Scripts to replace the definitions (starting in the first column) with another one
  - And store the information in versions-YY.MM.txt and versions-contrib-YY.MM.txt.
- Scripts:
  - tools/upgrade-tc.py: CSCS script for toolchain updates
  - tools/upgrade-locals.lua: LUST script for dependencies
  - Use via aliases

Tips & Tricks

Tips&Tricks

WiP, will be based on questions also.