RADIUSS CI Documentation

Release 0.1

Adrien M. Bernede

Jan 26, 2023

User Documentation

1 Background and Motivation **3**

2 Overview **5**
2.1 User Guide 5
2.2 Developer Guide 16

Documentation of the CI infrastructure developed for RADIUSS projects.

RADIUSS CI is a sub-project from the RADIUSS initiative focusing on sharing resource and documentation regarding Continuous Integration among RADIUSS projects.

Note: LLNL's RADIUSS project (Rapid Application Development via an Institutional Universal Software Stack) aims to broaden usage across LLNL and the open source community of a set of libraries and tools used for HPC scientific application development.

CHAPTER 1

Background and Motivation

Projects belonging to the RADIUSS scope are targeting the same machines and use Spack as a packaging system. We want them to ensure they build with similar tool chains.

We designed an automated *CI infrastructure based on GitLab* that we meant to be universal enough to be shared among RADIUSS projects. This infrastructure involves *using Spack to setup the project dependencies and generate a configuration file*. This allows projects to easily *share the full context of their builds*. The project is then built and tested as usual and most of *the CI infrastructure is shared* to avoid duplication and ease the maintenance.

CHAPTER 2

Overview

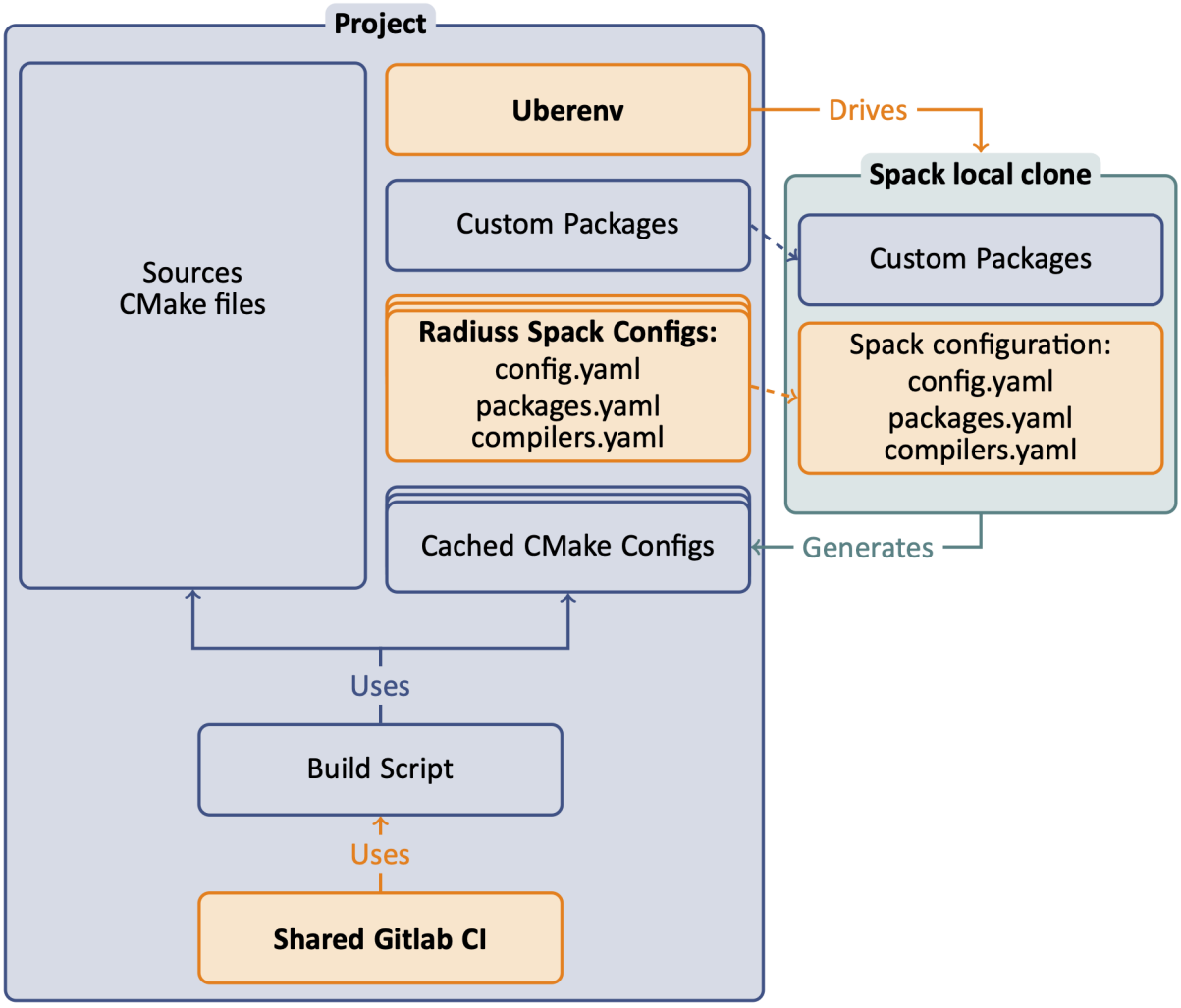
We split the design in three steps necessary to adopt RADIUSS CI methodology. Those actions will be documented in the **'user_guide'_**.

1. **Use Spack to configure the project build.** Spack provides a single context to express *toolchains*, *machines setup* and *build sequence*. It is increasingly used to install the dependency tree of large simulation codes.
2. **Build and test without breaking your habits.** We do not require the adoption of Spack to build your code but we require that your build system accepts the configuration file generated by Spack as an input (`CMakeCache.txt` for CMake build system). That way, dependencies and options are already set coherently with the spec provided to build the dependency tree.
3. **Setup the CI using the shared template.** Once you have put in the effort to adopt the first two steps, you should be able to benefit from the shared CI infrastructure. In very complex scenario, you would still be able to use the RADIUSS CI template as a starting point for a custom implementation.

In the **'dev_guide'_**, we discuss the layout of the RADIUSS CI infrastructure and how the different pieces work with one another. Technical choices are also explained there.

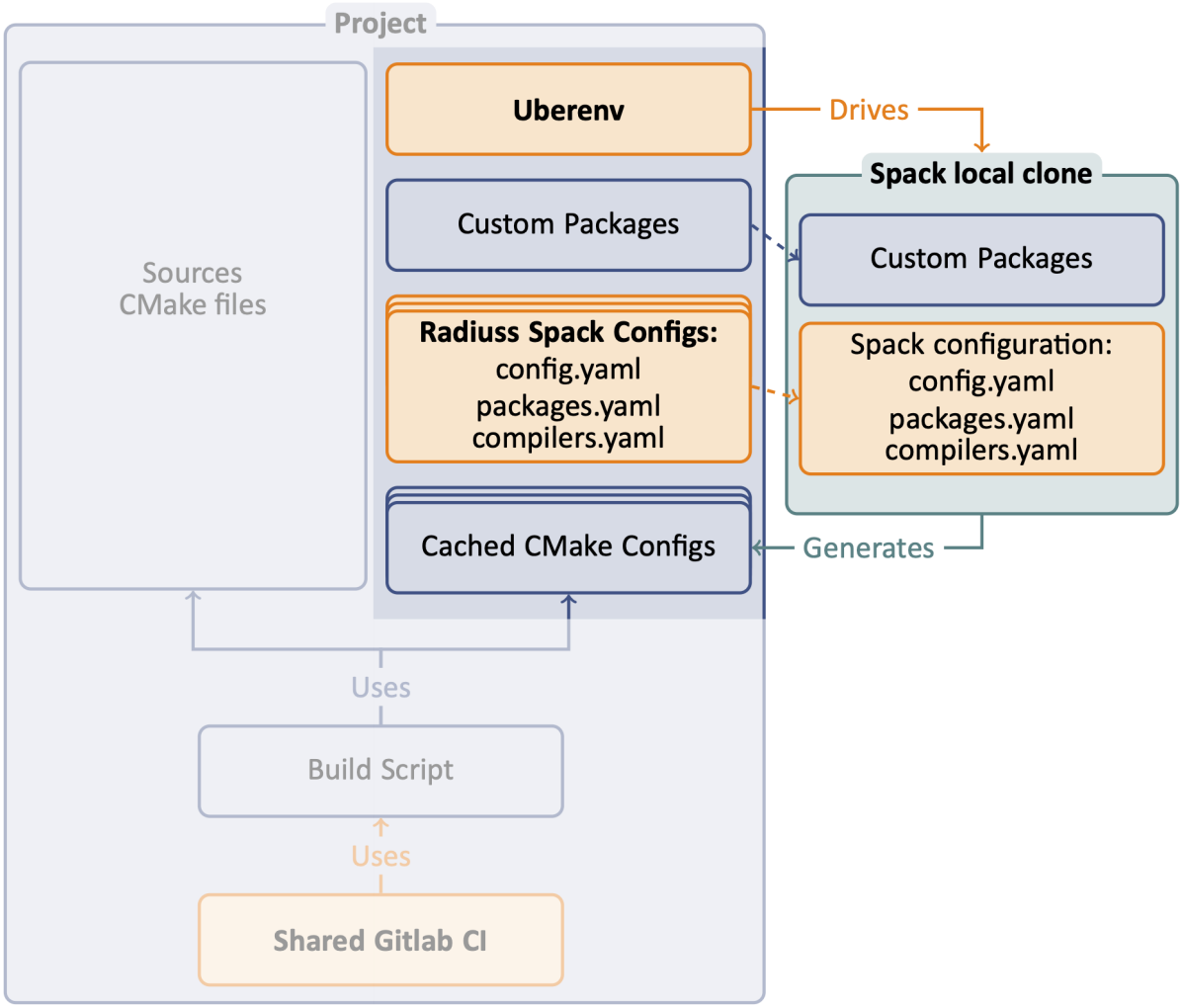
2.1 User Guide

We designed an automated *CI infrastructure based on GitLab* that we meant to be universal enough to be shared among RADIUSS projects. This infrastructure involves *using Spack to setup the project dependencies and generate a configuration file*. This allows projects to easily *share the full context of their builds*. The project is then built and tested as usual and most of *the CI infrastructure is shared* to avoid duplication and ease the maintenance.



We split the design in three steps necessary to adopt RADIUSS CI methodology.

2.1.1 Use Spack to configure the project build



The first step in adopting RADIUSS CI infrastructure is to setup your project so that Spack can be used to install the dependencies and generate a configuration file for the build.

The end product should be a script that takes a Spack spec as an input, and returns the configuration file generated by Spack after installing the dependencies for the given spec.

Spack provides a single context to express *toolchains*, *machines setup* and *build sequence*. Using it will allow us to share configuration files to describe the toolchains and machines setup. Radiuss-Spack-Configs is the repository where RADIUSS projects Spack configuration is shared.

Spack is increasingly used to install the dependency tree of large simulation codes. As such, it makes sense to use Spack early in the development process.

Note: We are not promoting a "Spack everywhere" strategy. But we advocate that Spack should be *one of the ways* to configure and build your projects, since you projects will likely be built that way in production someday.

We rely on Uberenv to facilitate the setup of a local and isolated spack instance that will be used to build the project dependencies. We strongly suggest that you start with Uberenv to benefit from a reliable Spack usage in your CI (tried

and tested) and keep the configuration script simple.

Uberenv Guide

The role of Uberenv will be to manage the setup of your Spack instance and then drive Spack to install your project dependencies and generate the configuration file.

Note: Uberenv will create a directory `uberenv_libs` containing a Spack instance with the required project dependencies installed. It then generates a CMake configuration file (`<config_dependent_name>.cmake`) at the root of the project repository.

One common source of error when using Uberenv is when the `uberenv_libs` folder is out of date. To resolve, make sure this folder is deleted before running new scripts for the first time because this folder needs to be regenerated.

Getting Uberenv by clone/fetch/copy

1. Get uberenv.py script.

   Clone/Fetch/Copy it from Uberenv repository. into a `uberenv` directory, not as a submodule.

2. Edit uberenv/project.json.

   Set your project package name, and other parameters like Spack reference commit/tag (we suggest the latest release tag).

3. Add radiuss-spack-configs submodule.

   - Use `git submodule add` to get **'Radiuss_Spack_Configs'_**.
   - Create a symlink `uberenv/spack_configs` that points to `radiuss-spack-configs`.

4. Add custom packages.

   If you need to make local modifications to your project package or a dependency package, you may put it in a corresponding directory:
   `uberenv/packages/<package_name>/package.py`.

5. Make sure that <project>/package.py generates a host-config cmake file.

   This is usually done adding a specific stage to the package (see for example the hostconfig stage in Umpire, CHAI, etc.).

Getting Uberenv as a submodule

1. Get uberenv.py script.

   Use `git submodule add` to get Uberenv into a `uberenv` directory.

2. Edit .uberenv.json.

   Create `.uberenv.json` in a directory that is a parent of `uberenv`. Set your project package name, and other parameters like Spack reference commit/tag (we suggest the latest release tag).

3. Add radiuss-spack-configs submodule.

   - Use `git submodule add` to get Radiuss-Spack-Configs in a second submodule or custom location.

- In `.uberenv.json` set `spack_configs_path` to point to `<some_path>/radiuss-spack-configs`.

4. Add custom packages.
   - If you need to make local modifications to your project package or a dependency package, you may put it in a corresponding directory:
     `<some_path>/packages/<package_name>/package.py`.
   - In `.uberenv.json` set `spack_packages_path` to point to `<some_path>/packages`
5. Make sure that <project>/package.py generates a host-config cmake file.

   This is usually done adding a specific stage to the package (see for example the hostconfig stage in Umpire, CHAI, etc.).

Get the shared Spack configuration

We share Spack configuration files in **'Radiuss_Spack_Configs'_**. In this repo you will find:

- *config.yaml* for Spack general configuration.
- *modules.yaml* for modules creation by Spack.
- One *compilers.yaml* and *packages.yaml* per system type, describing the installed toolchain on each machine.

Depending on the machine/system, we may or may not provide a spack configuration allowing you to use it right away. Please refer to **'Radiuss_Spack_Configs'_** documentation about adding a new machine. This will be welcome by the RADIUSS teams using it!

Note: MacOS (darwin) case It is not trivial to provide a universal configuration for MacOS. Instead, the developer will likely have to complete the `packages.yaml` file in order to adapt the location and version of externally installed dependencies. MacOS is not available on LC systems, the Spack configuration is provided as-is, for development use.

Setup your Spack package to generate a configuration file

We want to build the dependencies with Spack and then build the project with those dependencies but outside of Spack. We need to generate a CMake configuration file that reproduces the configuration Spack would have generated in the same context. It should contain all the information necessary to build your project with the described toolchain and dependencies.

In particular, the configuration file should setup:

- flags corresponding with the target required (Release, Debug).
- compilers path, and other toolkits (cuda if required), etc.
- paths to installed dependencies.
- any option that may have an impact on your build.

This provides an easy way to build your project based on Spack configuration while only using CMake and a traditionnal developer workflow.

CMake projects: Spack CachedCMakePackage

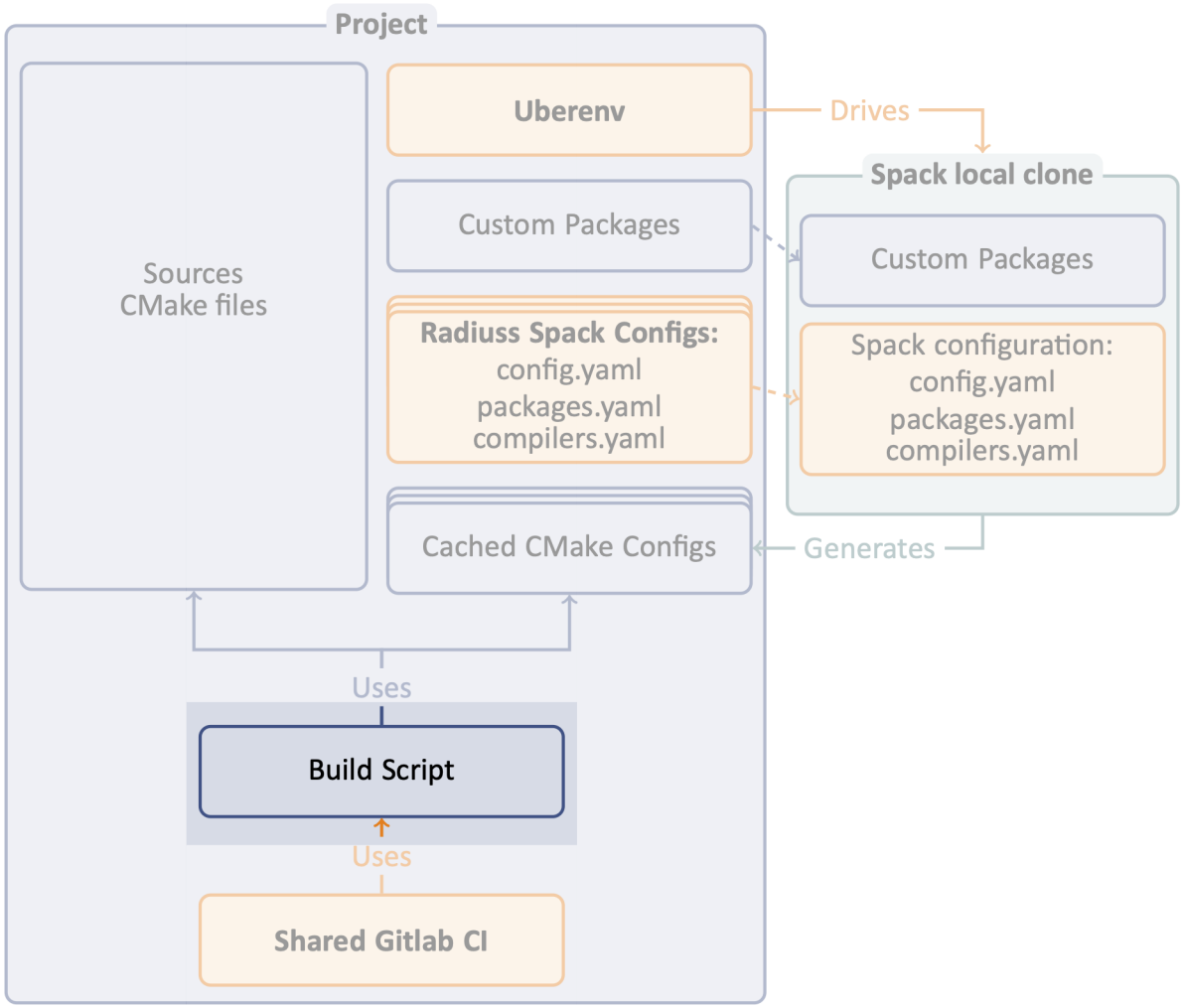
The use of a CMake build system is strongly recommended to adopt RADIUSS CI workflow, that's because of this step. With CMake, we can generate a cache file with all the configuration necessary to trigger a build later on. This is supported in Spack as soon as your package inherits from `CachedCMakePackage`.

Once your package has been ported, stopping the Spack install after `initconfig` phase will prevent it from building your project and the CMake configuration file will have been generated already.

Non-CMake projects: Custom implementation

The only example of a non-CMake project that adopted this workflow is MFEM. Altough it is using a Makefile build system in its Spack Packages, MFEM is generating a configuration file that can be used just like a CMake configuraton file. We adapted the implementation of the package to mimics the mechanism available in CMake-based packages. You may use that as an example.

2.1.2 Build and test without breaking your habits



The second step in adopting RADIUSS CI infrastructure is to make sure your project can be built using the configuration file generated by Spack. Other than that, building and testing your code should follow the usual development workflow.

Spack is not longer involved at this point. But using the configuration file will make sure the build uses the Spack installed dependencies and the options specified by the Spack spec.

Using configuration files to build the project

The (CMake) configuration files are specific to the desired machine and toolchain. With CMake, the usage is as follow:

```
$ mkdir build && cd build
$ cmake -C  <path-to>/<configuration>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

In the end, this should not be a major change in the developers habit: this is standard CMake procedure.

Writing a script for CI

The CI expects a script that:

- is named `scripts/gitlab/build-and-test`.
- is parametrized by the variable `SPEC` which should contain a Spack spec with the project name stripped out.
- covers both step 1 (installation of dependencies, configuration file generation) and step 2 (build the project from the configuration file, test it).

The script should therefore be calleable that way:

```
$ SPEC="%clang@9.0.0 +cuda" scripts/gitlab/build_and_test.sh
```

Note: Making the CI scripts usable outside CI context is recommended since, by definition, it has been vetted. It also ensures that this script is usable in interactive mode, making it easier to test. This is why to document it in the build part rather than the CI part.

Umpire, RAJA, CHAI, MFEM each have their own script you could easily adapt. All these projects use Uberenv to drive Spack. Umpire, RAJA and CHAI share the Spack configuration files in **'Radiuss-Spack-Configs'_** in order to keep building with the same tool-chains.

Debugging

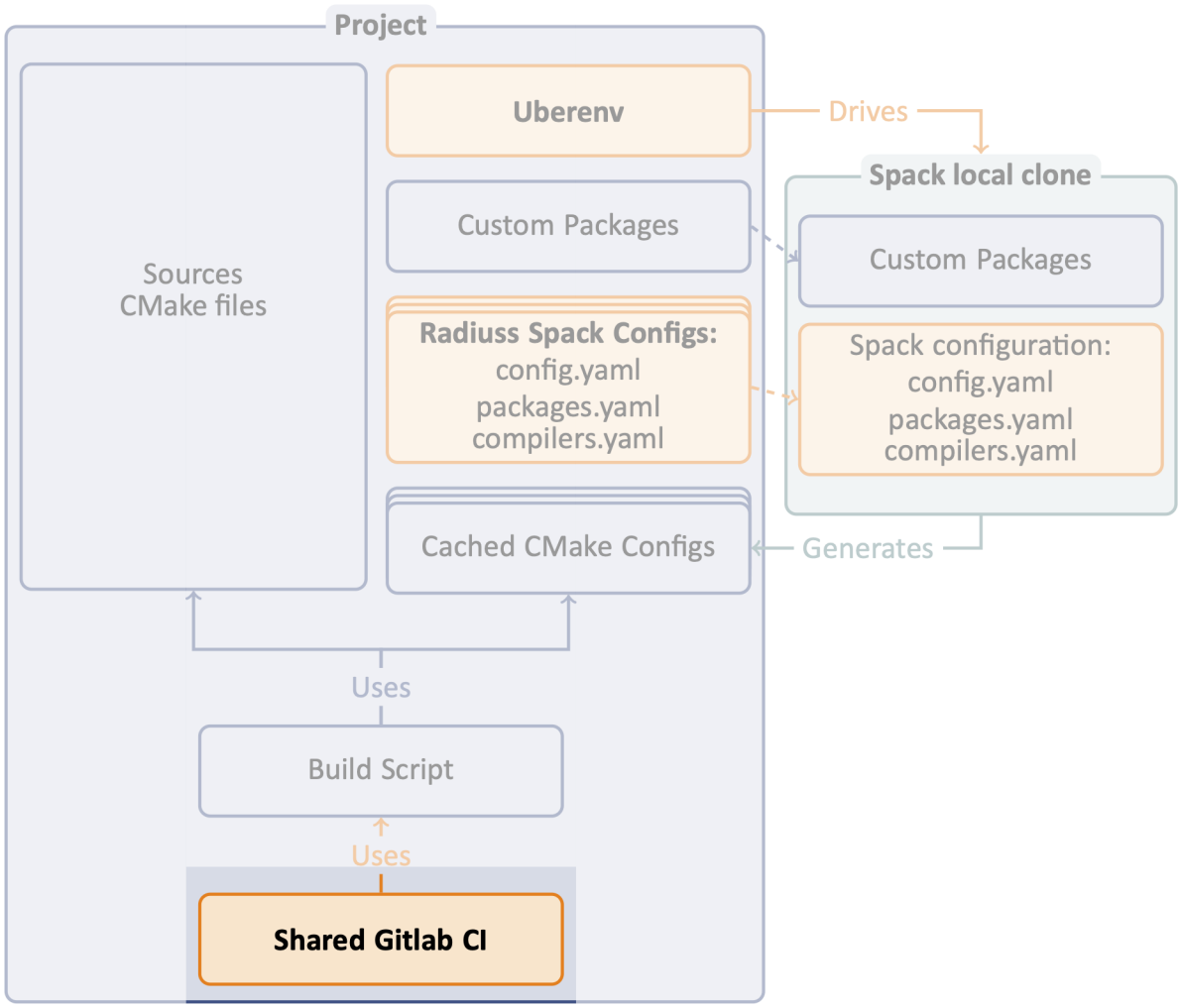
In the workflow described above, there are 4 levels of scripts to control the build of a package. From the lower to the higher level:

- The *build system* is controlled by the configuration file (generated by Spack or not).
- The *Spack package* is controlled by the spec provided and spack configuration.
- *Uberenv* takes a spec and a json configuration file.
- A *build_and_test script* also sometimes called test driver. The one in Umpire and RAJA requires a spec and some other control variables.

Now, when it comes to debugging, each level has some requirements to reproduce a failing build:

- The *build_and_test script* typically runs in CI context. This means that it may not be designed to run outside CI. It is better if it does, and we try to do that in RADIUSS, but it is not guaranteed. * *Uberenv* is meant to provide a turnkey way to install the project and its dependencies. It is usually a good way to reproduce a build on the same machine. The CI creates working directories in which the uberenv install directory _may_ persist, but it is better to reproduce in a local clone.
- Reproducing a build with *Spack* requires a deep knowledge of it. But Uberenv helps a lot with that. We advice that you use Uberenv to generate the Spack instance. Then, loading the spack instance generated and working with it is safe.
- Going down to the *build system* is also doable, especially when using the generated configuration files. Once spack has installed the dependencies and generated the configuration files, the latter can be used to control the build of the code and this should not require using Spack.

2.1.3 Setup the CI using the shared template



The third step in adopting RADIUSS CI infrastructure is to setup the CI.

Once you put in the effort to adopt the first two steps, you should be able to benefit from the shared CI infrastructure. In very complex scenario, you will always be able to use the template as a starting point for a custom implementation.

By sharing the CI definition, projects share the burden of maintaining it. In addition, with our shared CI, they also share a core set of toolchains (spack specs) to ensure that they keep running tests with similar configurations.

Radiuss Shared CI

Sharing the CI framework started with sharing spack configuration files, a method manage Spack and use it to generate CMake configuration files and a `build-and-test` script with that has the same imputs across projects. We will now also share most of the CI implementation itself.

By externalizing the CI configuration, we create the need for an interface. We try to keep this interface minimalistic, while allowing customization.

Note: GitLab allows projects to include external files to configure their CI. We rely on this mechanism to share most of the CI configuration among projects.

The short version

```
### Prerequisites
cd my_project
mkdir -p scripts/gitlab
vim scripts/gitlab/build-and-test
# write CI script

### CI Setup
cd my_project/..
git clone https://github.com/LLNL/radiuss-shared-ci.git
cd my_project
cp ../radiuss-shared-ci/customization/gitlab-ci.yml .gitlab-ci.yml
mkdir -p .gitlab
cp ../radiuss-shared-ci/customization/custom-*.yml .gitlab
cp ../radiuss-shared-ci/example-extra-jobs/*-extra.yml .gitlab
vim .gitlab/custom-*.yml
# customize CI
vim .gitlab/*-extra.yml
# edit extra jobs
```

Jump to the corresponding section to deal with *Customize CI*, *Edit extra jobs* and *Writing a script for CI*.

The detailed version

Our CI implementation can be divided in four parts:

- local build-and-test script
- shared files
- customization files
- extra jobs

Setting up the CI will basically consist in four corresponding phases.

Write CI Script

The very first step is to provide a CI script. You should already have one after completing **'write-ci-script'_** at Step 2.

Once you have that script, you can move on to the CI setup.

Core CI implementation

Start by cloning the project locally, for example next to the project you intend to add CI to.

```
cd my_project/..
git clone https://github.com/LLNL/radiuss-shared-ci.git
cd my_project
```

By default, GitLab expects a `.gitlab-ci-yml` file to interpret the CI setup. We provide one in `customization/gitlab-ci.yml` that projects can copy-paste, just be sure to place it at the root of your project, with a dot (`.`) at the beginning of the name.

```
cp ../radiuss-shared-ci/customization/gitlab-ci.yml .gitlab-ci.yml
```

Your CI is now setup to include remote files from the GitLab mirror of Radiuss-Shared-CI.

We now have to complete the interface with the shared CI config. Indeed, `.gitlab-ci.yml` also expects some files to be present locally. Those are the next steps.

Customize CI

We provide templates for the required customization files. We need to copy them in the `.gitlab` directory.

```
mkdir -p .gitlab
cp ../radiuss-shared-ci/customization/custom-*.yml .gitlab
```

We will now browse the files to see what changes they may require to suit your needs.

`.gitlab/custom-pipelines.yml`

In this file, you will select the machines you want to run tests on. Comment the jobs (sections) corresponding to machines you don't want, or don't have access to.

`.gitlab/custom-jobs.yml`

No change is strictly required to get started here.

In this file, you may add configuration to the `.custom_build_and_test` job that will then be included to all you CI jobs. This can be used for example to export jUnit test reports.

`.gitlab/custom-variables.yml`

We should now have a look at `.gitlab/custom-variables.yml`. Here is a table to describe each variable present in the file. Some more details can be found in the file itself.

Parameter	Description
LLNL_SERVICE_USER	Service Account used in CI
CUSTOM_CI_BUILD_DIR	Where to locate build directories (prevent overquota)
GIT_SUBMODULES_STRATEGY	Controls strategy for the clone performed by GitLab. Consider `recursive` if you have submodules, otherwise comment it.
BUILD_ROOT	Location (path) where the projects should be built. We provide a sensible default.
ALLOC_NAME	Name of the shared allocation. Should be unique, our default should be fine.
<MACHINE>_BUILD_AND_TEST_SHARED_ALLOC	Parameters for the shared allocation. You may extend the resource and time.
<MACHINE>_BUILD_AND_TEST_JOB_ALLOC	Parameters for the job allocation. You may extend the resource and time within the scope of the shared allocation.

Note: If a variable is blank in the template file, then it does not require a value. If a variable has a value there, it does require one.

Warning: We strongly recommend that you set your CI to use a service account.

Edit extra jobs

We provide templates for the extra jobs files. Those files are required as soon as the associated machine has been activated in `.gitlab/custom-pipelines`.

If no extra-jobs is needed (the shared jobs are automatically included), then you should add the extra-jobs files as-is, with a simple variable definition to avoid it to be empty.

If you need to define extra-jobs specific to your projects, then you may remove the variable definition, uncomment the template job and complete it with the required information:

- A job name, unique, that will appear in CI.
- A Spack spec used by `build-and-test` to know what to build.

Note: Gitlab supports long and complex job names. Make sure to pick a unique name not to override a shared job.

We also provide an "How To" section.

2.1.4 How To

List the Spack specs tested

RADIUSS Shared CI uses Spack specs to express the types of builds that should be tested. We aim at sharing those specs so that projects build with similar configurations. However we allow projects to add extra specs to test locally.

Shared specs for machine `ruby` can be listed directly in Radiuss-Shared-CI:

```
cd radiuss-shared-ci
git grep SPEC ruby-build-and-test.yml
```

Extra `ruby` specs, specific to one project, are defined locally to the project in `.gitlab/ruby-build-and-test-extra.yml`

```
cd <project>
git grep SPEC .gitlab/ruby-build-and-test-extra.yml
```

Use Uberenv

```
$ ./scripts/uberenv/uberenv.py
```

Note: On LC machines, it is good practice to do the build step in parallel on a compute node. Here is an example command: `srun -ppdebug -N1 --exclusive ./scripts/uberenv/uberenv.py`

Unless otherwise specified Spack will default to a compiler. It is recommended to specify which compiler to use: add the compiler spec to the `--spec=` Uberenv command line option.

Some options

`--spec=` is used to define how your project will be built. It should be the same as a spack spec, without the project name:

- `--spec=%clang@9.0.0`
- `--spec=%clang@8.0.1+cuda`

The directory that will hold the Spack instance and the installations can also be customized with `--prefix=`:

- `--prefix=<Path to uberenv build directory (defaults to ./uberenv_libs)>`

Building dependencies can take a long time. If you already have a Spack instance you would like to reuse (in supplement of the local one managed by Uberenv), you can do so with the `--upstream=` option:

- `--upstream=<path_to_my_spack>/opt/spack ...`

Warning: Due to its GitLab CI sharing goal, Radiuss Shared CI is meant to live on LC GitLab instance. The main repo, hosted on GitHub for accessibility and visibility, is mirrored on LC GitLab. To include files from Radiuss-CI, we recommend pointing to the mirror repo on GitLab rather than the GitHub one. We only document that option.

2.2 Developer Guide

There should be two types of contributions to this Radiuss-Shared-CI: adding new shared jobs, contributing changes to the CI implementation (and documentation).

We provide guidance for both in this section, using an HowTo format where we describe the set of actions to perform for several practical use-cases.

This section of the documentation also explains some technical choice.

2.2.1 How To

Add a new machine

Adding a new machine can be done directly in this project so that the configuration is shared with all. However it the associated Spack configuration must first be added to Radiuss-Spack-Configs.

The sub-pipeline definition

To add a new machine, create a corresponding `<machine>-build-and-test.yml` file describing the sub-pipeline. There are two main cases: whether the machine uses Slurm or LSF Spectrum as a scheduler.

Machines using Slurm scheduler

For machines using Slurm scheduler, use `ruby` (or `corona`) as a starting point. Then replace all the instances of "RUBY" and "ruby" with the new machine name.

Then go to `customization/custom-variables.yml` and add the variables:

- `<MACHINE>_BUILD_AND_TEST_SHARED_ALLOC` with allocation options sufficient for the shared allocation (salloc) to contain all the jobs.
- `<MACHINE>_BUILD_AND_TEST_JOB_ALLOC` with allocation options for any of the jobs (srun) the machine will take. The job will run under the shared allocation, also, do not prescribe a number of cores here, as they should be defined at Spack and Make/CMake level.

Note: Use the values we have for ruby and corona as guides, but adapt the partition, number of cpus per task and duration coherently with the machine.

Machines using LSF Spectrum scheduler

For machines using LSF Spectrum scheduler, use `lassen` as a starting point. Then replace all the instances of "LASSEN" and "lassen" with the new machine name.

Then go to `customization/custom-variables.yml` and add the variables:

- `<MACHINE>_BUILD_AND_TEST_JOB_ALLOC` with allocation options for any of the jobs the machine will take.

Note: Use the values we have for lassen as guides, but adapt the partition and duration coherently with the machine.

Reference the new sub-pipeline

In `customization/custom-pipelines.yml`, add a new section corresponding to the new machine. This is used by `customization/gitlab-ci.yml` to control which sub-pipelines are effectively generated.

Changelog

Don't forget to provide a quick description of your changes in the `CHANGELOG.md`.

New tag

Once the new machine setup is tested and valid, submit a PR. We will review it and merge it. We may create a new tag although it is not required for a new machine. Indeed, using a new machine is a voluntary change for users: they will have to activate it in `customization/custom-pipelines.yml` the same way you did above (which is a suggested template).

2.2.2 RADIUSS Shared CI explained

Radiuss-Shared-CI is an infrastructure and documentation repository created to help RADIUSS projects adopt the Gitlab CI workflow designed for them.

Project structure

Shared CI files

This project hosts the shared CI configuration, which can be found in the YAML files at the root of the project: `<machine>-build-and-test.yml`.

Each file contains both the configuration and the jobs for one machine. They assume that some entries will be provided by including the customization files.

Customization files

The `customization` directory holds all the files allowing to customize the pipeline.

The files `custom-pipelines.yml`, `custom-variables` and `custom-jobs.yml` are all included in the configuration in `gitlab-ci.yml`.

Project must use `gitlab-ci.yml` as a base for the `.gitlab-ci.yml` at the root of their project. This files does not require any change, but can receive additional stages if needed by the project.

Note: We could share `.gitlab-ci.yml` directly in Radiuss-Shared-CI. However that would require projects to configure GitLab, through the UI, to use that file. This is not considered a good idea at the moment, and we preffer projects to have the capability to easily add other stages to their CI configuration.

Both `custom-pipelines.yml` and `custom-variablse.yml` are included globally. They will affect all the CI workflow. The file `custom-jobs.yml` is included in the `build-and-test` sub-pipelines and will only affect those ones.

Extra jobs

Extra jobs can be defined by the user appending `.gitlab/${CI_MACHINE}-build-and-test-extra.yml`. We provide a working minimal template that should always be provided.

Note: Strictly speaking, the extra jobs file is only needed once the associated sub-pipeline is reference to `.gitlab-ci.yml` through `customization/custom-pipelines.yml`. We make it presence mandatory for the sake of simplicity.

Other files

The documentation source code is in the `docs` directory, while `cmake` aims at receiving BLT submodule to manage the local build of the docs.

References

LC specific documentation for Gitlab. In particular, the "Getting Started" and "Setup Mirroring" sub-pages.