

A little disclaimer:

This is an "internal" type of instructions document, somewhat informal, because I have found myself in a position to instruct people on things that I am not necessarily comfortable with. In any case, if you have problems with any of the below, contact me at marta.pienkowska@erdw.ethz.ch and I will do my best to help :-)

Good luck,
Marta

1. Installation

1- Dependencies

First, you have to install docker and docker compose (docker engine)

<https://docs.docker.com/install/linux/docker-ce/ubuntu/>

Then make sure you can run docker without `sudo`:

<https://docs.docker.com/engine/install/linux-postinstall/>

If you cannot or do not want to run without `sudo`, you will have to run with the `-E` flag to make sure that you preserve the environment variables (i.e. all `docker compose` commands below have to be preceded with `sudo -E`).

Note: That's not exactly a dependency, but you also need to have the SSH keys set up for the machine where you will be running simulations and also the `SSH_AUTH_SOCK` environment variable needs to be defined (should be automatic, but it turns out that it does not work for all Linux distributions).

2 - UCIS4EQ services

1. Because the Docker images are registered on a GitLab repository, first you have to do the login. `-p` specifies the password, and it is the project access token for a private project. It will ask you to provide your GitLab username. This is not the e-mail address that you use for login, but the actual `@username`.

```
docker login registry.gitlab.com -p glpat-2fr-PF8pNpCErgeTyR31
```

2. Then create a UCIS4EQ working directory, move there and run:

```
bash -c "$(curl --request GET --header 'PRIVATE-TOKEN: glpat-75-FV88yWpzcZJvoKfe'
'https://gitlab.com/api/v4/projects/43953314/repository/files/install.sh/raw?ref=marta_devel')"
```

This downloads and runs the `install.sh` script from a private repo.

3. Set your own credentials in `DAL.json`:
 1. For BSC_B2DROP, change just `user` and `pass` fields.
 2. If you are on Marenosturm4, change the `user` and the `path`. The `path` is the folder for runs and is probably going to be in your `scratch`.
 3. If you are on Daint, change the `user`, the `proxy` and the `path`. The `path` is the folder for runs and is probably going to be in your `scratch`.
4. Set your own credentials in `Sites.json`:
 1. If you're on Daint, just keep Daint and remove MN4, and vice versa. There is some sort of priority system for which machine to choose if both are defined, but it is just a manually set index for now (in one of the config files that are on B2DROP, so it is *the same for all users...* not ideal...). So to avoid confusion, just keep a single one.
 2. Set your `user` and your `path` again (make it the same as in `DAL.json` - I think one of those is just a legacy thing and is not read, but I actually do not know which one that is), and the `proxy` if on Daint.
 3. Then there are the resources. They are set for the small test case that we are going to run. This is some sort of temporary solution that was set up before a proper resource allocation is in place - it is far from ideal, as you have to adjust this now for every test case (that is, mostly the `wtime`, `nodes`, and `tasks-per-node` for the `SalvusRun` itself).
5. To run you have to make sure that the machine/cluster you're meaning to work on has the ssh key set up with the localhost and that it has been added to the `~/.ssh/known_hosts` file. Also check that the `SSH_AUTH_SOCK` variable is defined (run `echo $SSH_AUTH_SOCK` and see if it outputs something) - it should be automatically defined if you have an SSH agent installed, but it turns out that on some distributions it is not and you have to do it yourself (e.g. Arch Linux).

When you run for the first time, run:

```
docker compose up
```

It will download the images and run all the services. Note that depending on your docker compose installation your command might be `docker-compose` (rather than `docker compose`).

For running in a "regular" way, I recommend that you run in two separate terminal windows:

```
docker compose up slipgen salvus microservices
```

and

```
docker compose up workflowmanager
```



This way you have a better overview of what is happening in the services. If you run all together, the `workflowmanager` output will dominate and it is harder to find your way through the on-screen log.

If you have set everything up correctly and run the two `docker compose up` commands above, the outputs should look like this:

```
File Edit View Search Terminal Help
marta@mp:~/Install/devel/ucis4eq_devel/ucis4eq$ docker compose up salvus slipgen microservices
WARN[0000] The "LOCATION" variable is not set. Defaulting to a blank string.
[+] Running 5/5
  :: Network ucis4eq_default                Created
  :: Container ucis4eq-DAL-1                 Created
  :: Container ucis4eq-salvus-1              Created
  :: Container ucis4eq-microservices-1      Created
  :: Container ucis4eq-slipgen-1            Created
Attaching to ucis4eq-microservices-1, ucis4eq-salvus-1, ucis4eq-slipgen-1
ucis4eq-slipgen-1 | * Serving Flask app 'slipGenService'
ucis4eq-slipgen-1 | * Debug mode: on
ucis4eq-slipgen-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
ucis4eq-slipgen-1 | * Running on all addresses (0.0.0.0)
ucis4eq-slipgen-1 | * Running on http://127.0.0.1:5002
ucis4eq-slipgen-1 | * Running on http://192.33.105.133:5002
ucis4eq-slipgen-1 | Press CTRL+C to quit
ucis4eq-slipgen-1 | * Restarting with stat
ucis4eq-salvus-1 | * Serving Flask app 'salvusService'
ucis4eq-salvus-1 | * Debug mode: on
ucis4eq-salvus-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
ucis4eq-salvus-1 | * Running on all addresses (0.0.0.0)
ucis4eq-salvus-1 | * Running on http://127.0.0.1:5003
ucis4eq-salvus-1 | * Running on http://192.33.105.133:5003
ucis4eq-salvus-1 | Press CTRL+C to quit
ucis4eq-salvus-1 | * Restarting with stat
ucis4eq-slipgen-1 | * Debugger is active!
ucis4eq-slipgen-1 | * Debugger PIN: 481-254-029
ucis4eq-salvus-1 | * Debugger is active!
ucis4eq-salvus-1 | * Debugger PIN: 128-743-119
ucis4eq-microservices-1 | * Serving Flask app 'microServices'
ucis4eq-microservices-1 | * Debug mode: on
ucis4eq-microservices-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
ucis4eq-microservices-1 | * Running on all addresses (0.0.0.0)
ucis4eq-microservices-1 | * Running on http://127.0.0.1:5000
ucis4eq-microservices-1 | * Running on http://192.33.105.133:5000
ucis4eq-microservices-1 | Press CTRL+C to quit
ucis4eq-microservices-1 | * Restarting with stat
ucis4eq-microservices-1 | * Debugger is active!
ucis4eq-microservices-1 | * Debugger PIN: 139-129-556

ucis4eq-workflowmanager-1 | WARNING: Unexpected argument: resource Found in @http decorator.
ucis4eq-workflowmanager-1 | WARNING: Unexpected argument: service_name Found in @http decorator.
ucis4eq-workflowmanager-1 | WARNING: Unexpected argument: request Found in @http decorator.
ucis4eq-workflowmanager-1 | WARNING: Unexpected argument: resource Found in @http decorator.
ucis4eq-workflowmanager-1 | WARNING: Unexpected argument: service_name Found in @http decorator.
ucis4eq-workflowmanager-1 | Binding debug is activated
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_On - Creating the JVM
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @create_vm - reading file in JVM_OPTIONS_FILE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @create_vm - Launching JVM
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @create_vm - JVM Ready
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_On - Obtaining Runtime classes
ucis4eq-workflowmanager-1 | WARNING: COMPSs Properties file is null. Setting default values
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_On - Creating runtime object
ucis4eq-workflowmanager-1 | [[536] API] - Deploying COMPSs Runtime v3.0.rc2210 (build 20221024-1010.rc66ec82ef61a7a2666f37b5a5d9ff3bfe7e02677)
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_On - Calling runtime start
ucis4eq-workflowmanager-1 | [[536] API] - Starting COMPSs Runtime v3.0.rc2210 (build 20221024-1010.rc66ec82ef61a7a2666f37b5a5d9ff3bfe7e02677)
ucis4eq-workflowmanager-1 | [[536] API] - Initializing components
ucis4eq-workflowmanager-1 | [[861] API] - Ready to process tasks
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Types
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Types DONE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Master
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Methods
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Methods DONE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI OnFailure Types
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Direction Types
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Direction Types DONE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Stream Types
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Stream Types
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Parameter Prefix
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init JNI Parameter Prefix DONE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @Init Master DONE
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_Get_AppDir - Getting application directory.
ucis4eq-workflowmanager-1 | [BINDING-COMMONS] - @JNI_Get_AppDir - directory name: /root/.COMPSs/workflowManagerService.py_01/
ucis4eq-workflowmanager-1 | * Serving Flask app 'launch'
ucis4eq-workflowmanager-1 | * Debug mode: off
ucis4eq-workflowmanager-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
ucis4eq-workflowmanager-1 | * Running on all addresses (0.0.0.0)
ucis4eq-workflowmanager-1 | * Running on http://127.0.0.1:5001
ucis4eq-workflowmanager-1 | * Running on http://192.33.105.133:5001
ucis4eq-workflowmanager-1 | Press CTRL+C to quit
```

Note that this does not start up the `listener` service for automatic submissions, but just the containers needed for manual submissions. If you want to see the `listener` in action, start it up (in a separate terminal window) with

```
docker compose up listener
```

6. To stop running the services hit `ctrl+C` twice in all terminal windows where services are running, and to properly clean up run (in just one window):

```
docker compose down
```

Note: an alternative way of adding SSH keys to the dockers

If things worked for you and your SSH connection works when you run, you can ignore this. :-)

Some of the services need to be able to establish an SSH connection to the cluster, so the docker images need to have the ssh key set up. The docker images that are pulled from the container registry do not have any ssh keys, and by running the `docker compose up` command you use the `docker-compose.yml` file which mounts two things from the localhost to the containers at runtime:

1. The `SSH_AUTH_SOCK` environment variable.
2. The `~/.ssh/known_hosts` file.

This should work on a Linux machine and on a Mac one (tested on both). I am not sure if this works on a Windows machine and the internet suggests that it may not. In any case, **this method is safe, as it does not permanently add the SSH keys anywhere and just mounts the local ones at runtime.**

If for some reason it does not work, you can use another deployment option which builds an extra layer to the images that require the SSH connection (this is the original way in which Juan Esteban set this up). Those images are local and you should be **careful to not upload them anywhere, as they contain your private SSH key!**

If you are on Daint run (where `PD_USER` is your Piz Daint username):

```
SSH_PRV="$(cat ~/.ssh/private_key_name)" PD_USER="username" docker compose -f docker-compose-add-ssh-key-to-docker.yml up
```

or if you are on MN4 run:

```
SSH_PRV="$(cat ~/.ssh/private_key_name)" docker compose -f docker-compose-add-ssh-key-to-docker.yml up
```

Then, for all subsequent runs, you can do (as above) in two separate terminal windows:

```
docker compose -f docker-compose-add-ssh-key-to-docker.yml up slipgen salvus microservices
```

and

```
docker compose -f docker-compose-add-ssh-key-to-docker.yml up workflowmanager
```

This will run the re-built images instead of the basic ones which mount the ssh information at runtime.

3 - Setting up on a new machine

The above assumes that you are using an installation on the HPC cluster that already is there and ready.

If this is not the case, a new cluster needs to be set up.

On the cluster itself:

1. `slipgen.simg`
2. `salvus_urgent_wrapper` (private git repo) + python environment with dependencies
3. `data` folder with regions (currently region = use case; later ideally regions will be significantly larger meshes that we will mask for simulations); in each folder there are four subfolders: `BATHYMETRY` , `MESHES` , `TOPOGRAPHY` , `VELOCITY_MODEL`

In the code:

1. need to add a `machinenameRunner.py` script with the setup of preparing the submission scripts in `ucis4eq/code/components/launchers`

In the setup:

1. add machine to `Sites.json` and `DAL.json` (local machine)
2. add machine to B2DROP in:
 - `UCIS4EQ/Setup/DALv2/Repositories.json`
 - `UCIS4EQ/Setup/DALv2/Resources.json`

ToDo: need to write this up better.

4 - Only for developments: running in debug mode and building new images

This might be trivial to some, but I am including this as a note for future reference for the "scientific" developers like me who do science and try their best to code, but get demoralised when confronted with proper code and proper software developers ;-)

4.1 Debug mode

If you have access to the source code and you have changed something, you can first test it without the necessity of re-building the images.

If you run in two separate terminal windows:

```
docker compose -f docker-compose-debug.yml up slipgen salvus microservices
```

and

```
docker compose -f docker-compose-debug.yml up workflowmanager
```

you are going to mount the local code (with the changes you have implemented) into the docker containers at runtime. This means that the old UCIS4EQ that was included in the docker images gets overwritten with your local version. If you want to check if the debugging mode worked as expected, you can run `docker ps` to list the running containers (after you have put up the services with the `up` commands above) and then enter the container to explore what is inside by running:

```
docker exec -it container-name bash
```

You can see the `container-name` in the output of the `docker ps` command.

4.2 Rebuild images

If you are then sure that you want to include that addition into the images, you can rebuild them. I would first run:

```
docker images
```

to list the images and see their IDs. Then:

```
docker rmi -f imageID1 imageID2 ...
```

where imageIDs are the IDs of the UCIS4EQ images.

This way you remove the old images (I am not sure if this is absolutely necessary or not, but I was doing this to be on the safe side), and then you can rebuild them by running:

```
docker compose -f docker-compose-build.yml build
```

Note that this is going to take a while (~2h or so).

Finally you have to push the new containers to the repository:

```
docker compose -f docker-compose-build.yml push
```

This assumes that you are logged in via `docker login registry.gitlab.com` and that you have the rights to push images to the repo.