**lwfm – Local Workflow Manager**

Programmer Guide

Updated: September 27, 2022

Table of Contents

[Overview & Architecture 1](#_Toc115205371)

[Major Objects 4](#_Toc115205372)

[How to Run lwfm 5](#_Toc115205373)

[Extending lwfm 5](#_Toc115205374)

[~/.lwfm/sites.txt 6](#_Toc115205375)

[Programming Workflows in lwfm 6](#_Toc115205376)

# Overview & Architecture

lwfm, the “local workflow manager”, is an implementation of a “type 3” workflow tool, as described in our paper.[[1]](#footnote-1)

* Type 1: in-situ workflows, within a Job and/or resource allocation (e.g., a coordinated multi-app / MPMD / multi-physics workflow running within a single user’s HPC Job allocation)[[2]](#footnote-2)
* Type 2: multi-Job workflows within a site / organization / enterprise (e.g., a typical HPC scheduler runs multiple Jobs on behalf of the same/different users concurrently and/or in sequence)
* Type 3: multi-Job / multi-site workflows (e.g., a MxN CI/CD system which builds and tests multiple applications on multiple target platforms with rollup reporting)

In order for *Sites* to interoperate in a workflow, we need to normalize their interface along four main functional aspects or subsystems:

* Auth: authenticate the user on the Site
* Run: run Jobs on the Site, get Job status, cancel Job
* Repo: put & get data to/from the Site, ideally using metadata decoration and search
* Spin: provision resources from a cafeteria of Site options, even from thin air (i.e., cloud) – spin up, spin down, track usage

Timeline

Description automatically generated

lwfm itself implements some middleware to orchestrate the type 3 workflows:

* A “local” runtime, allowing Jobs which run on the user’s own machine(s) to be 1st class citizens in the lwfm ecosystem. They run asynchronously.
* A Job Status Sentinel, which exposes a REST API for setting / unsetting Event Handlers which are keyed on Job status events. The JSS exposes endpoints for local Jobs to emit Job status, and for maintaining scheduled polling against remote Sites for the status of Jobs which have registered Event Handlers.
* A persistent store for Job status, aka the RunRepo.
* A persistent metadata store, aka the MetaRepo. This is capable of tracking and providing arbitrary metadata management for data objects which are stored locally (e.g., on a local filesystem), and which are put / get from remote Sites.

lwfm is based on several design perspectives and assumptions:

* RunRepo + MetaRepo = digital thread. Data is associated with the Job which created it and the Job(s) which used it. The thread is navigable from control flow or data flow perspectives.
* Workflows across Sites require a simplifying normalization of interfaces – standard ways to run Jobs, normalize Job status, get and put data. These simple interfaces must have wide open argument lists to handle the complexity inherent in the Site.
* Thus, there is no effort to hide the complexity of the Site – we intend for users to be able to use every Site-specific feature of, for example, that runtime in a heterogeneous ecosystem.
* lwfm is an interoperability framework, not a portability framework. There is no effort to author a workflow which can be deployed anywhere / run anywhere. We acknowledge the vast array of workflow tools, runtimes, and workflow encoding schemes, and encourage the user to use what they like, as an “application”, then weave these multi-Site workflows together as “the workflow is the app”.
* The perspective is “me and my data”. The point of view of the user is their own local workstation. From this vantage they orchestrate workflows across a number of system Sites to which they have access. (e.g., workflows which use local processes and enterprise workflow systems and external systems like those at national laboratory leadership computing centers).
* There is no assumption that a process on a remote Site can reach back to lwfm running locally. All communication is 1-way. Everything involving remote Sites involves polling, not callbacks.
* There is no assumption that we can deploy a lwfm service on a remote Site and have it accessible (e.g., via a REST call) to lwfm running on the user’s local machine. The Site exposes its runtime services through the Site Run interface only.

# Major Objects

The code implements several major objects, some of which get extended by implementors of new Site drivers.

Graphical user interface

Description automatically generated

* The Site interface consists of four sub-interfaces.
* A Site Driver, like the Local Site Driver, implements the sub-interfaces, or borrows them from other Site implementations.
* A Job Defn represents the description of a Job. Once submitted to the Site Run subsystem, it becomes a Job running inside a Job Context, and emits Job Status messages each associated with the same Job Context.
* Site File Ref abstracts “file” objects on remote Sites. Those might be files on actual filesystems, or some other more exotic form of object storage, perhaps one which also tracks object metadata.

# How to Run lwfm

To begin:

* You will need a recent Python 3 with the kitchen sink of Anaconda libraries.
* Set your Python classpath to include the lwfm src/ folder.
* Run the Job Status Sentinel service, which provides the event handling and logging

export FLASK\_APP=lwfm/server/JobStatusSentinelSvc

flask run -p 3000

You’re now ready to write and run lwfm workflows.

# Extending lwfm

The lwfm package ships with several Site drivers, each of which implements (or inherits the implementation of) each of the four subsystems of Auth, Run, Repo, and optionally Spin. To construct your own Site:

* Implement or extend a Site class with Auth, Run, Repo, and optionally Spin constituent classes. Implement the basic functionality for each of the exposed verbs. Throw a “not supported” exception for any methods which your Site doesn’t provide.
* Implement or extend a Job Status class which maps the job status terminology from your Site to the lwfm standard set of status names. For example, your Site might call a Job’s terminal state “done”, while lwfm calls it “COMPLETE” – you provide the mapping.

Typically, a Site is a computing resource – a named HPC cluster for example. A Site may contain more than one resource, exposed as *compute types* within it. Thus, when modeling your own Site with multiple resources within, you can choose to model it as one Site with multiple compute types, or many Sites with a shared implementation of one or more of Auth, Run, Repo, Spin.

And, while typically a Site is a computing resource, it need not be. Any networked device which one can envision in terms of the four subsystems can be modeled. For example, a large microscope – it may have some form of Auth, and it might make sense to “put” a configuration of a sampling or sampling series, “run” the job, and then “get” the results (images) from the device. In this context, Spin might for example alert a human-in-the-loop to prepare the device.

## ~/.lwfm/sites.txt

In addition to built-in drivers, lwfm will search a file ~/.lwfm/sites.txt for a   
“name=class” map. Here *name* is the name of the Site, and *class* is the name of the Python class on the classpath which implements the Site interface for that Site. Give your Site a name and map it to the Site class you provide.

# Programming Workflows in lwfm

lwfm ships with several example workflow scripts. A typical workflow will:

* Instantiate a Site driver by name

site = Site.getSiteInstanceFactory(siteName)

* Login to the Site

site.getAuthDriver().login()

* Construct a Job Definition

jobDefn = JobDefn()

jobDefn.setEntryPoint("echo 'hello world'")

* Submit the Job to run

status = site.getRunDriver().submitJob(jobDefn)

Note that Job execution is assumed to be asynchronous – e.g., submitted to a job scheduler for that Site. As a result, the status returned from the initial call to submit will likely return some initial pre-execution status code.

* Optionally wait for the Job to finish. The status contains a Job Context which includes the id of the Job – both the local lwfm Job id and the native Job id on the Site. Note that status codes are represented in their lwfm normalized form.

context = status.getJobContext()

status = site.getRunDriver().getJobStatus(context)

* Alternatively to polling on Job completion we can set an Event Handler - e.g., “when Job 123 running on Site ABC reaches state X, run a defined Job on Site PDQ”.

jssc = JobStatusSentinelClient()

jssc.setEventHandler(jobContextA.getId(), siteName,

JobStatusValues.COMPLETE.value,

jobDefnB, siteName, jobContextB)

* Data can be moved to/from the Site using the Repo interface methods. Data movement can be conveniently wrapped as a Job and executed or set as the target of an Event Handler.

jobDefnC = RepoJobDefn()

jobDefnC.setRepoOp(RepoOp.PUT)

jobDefnC.setLocalRef(Path(dataFile))

jobDefnC.setSiteRef(FSFileRef.siteFileRefFromPath(

os.path.expanduser('~')))

# Next Steps

The authors invite collaboration. The git project includes a todo.txt file serving as a rudimentary idea and issue tracker.

Some major areas for growth include:

* Formalisms for the MetaRepo, including editing, version tracking, etc. Use of a “real DB”, same for RunRepo (current demo implementation is file-based).
* Exposing a laboratory device (e.g., a microscope) as a Site and using it in workflows.
* Growing the set of Site drivers for major national systems.
* GUI for monitoring Job status.
* Graph-oriented visualization and navigation of the digital thread.
* Standard implementations of the Spin interface for major cloud providers, aggregators, and Software Defined Everything-ers.
* Full implementation of an Event Handler model with event groups, fuzzy and time-limited satisfaction, recurring and scheduled events, etc.

1. From the to-be published proceedings of the Oak Ridge National Lab Smoky Mountains Conference 2022, a pre-print available here: <https://drive.google.com/file/d/1c3YEVmEAUjbI5urj4PiV2TtjzBUzLlws/view?usp=sharing> [↑](#footnote-ref-1)
2. Nouns in capital letters are proper nouns. Job means “a Site runtime entity which has an id”. It might be a single process, or it might be thousands of processes distributed across an HPC cluster. It means what most people would normally think it means in the context of HPC computing. [↑](#footnote-ref-2)