JuliaRun Documentation

Hello, World

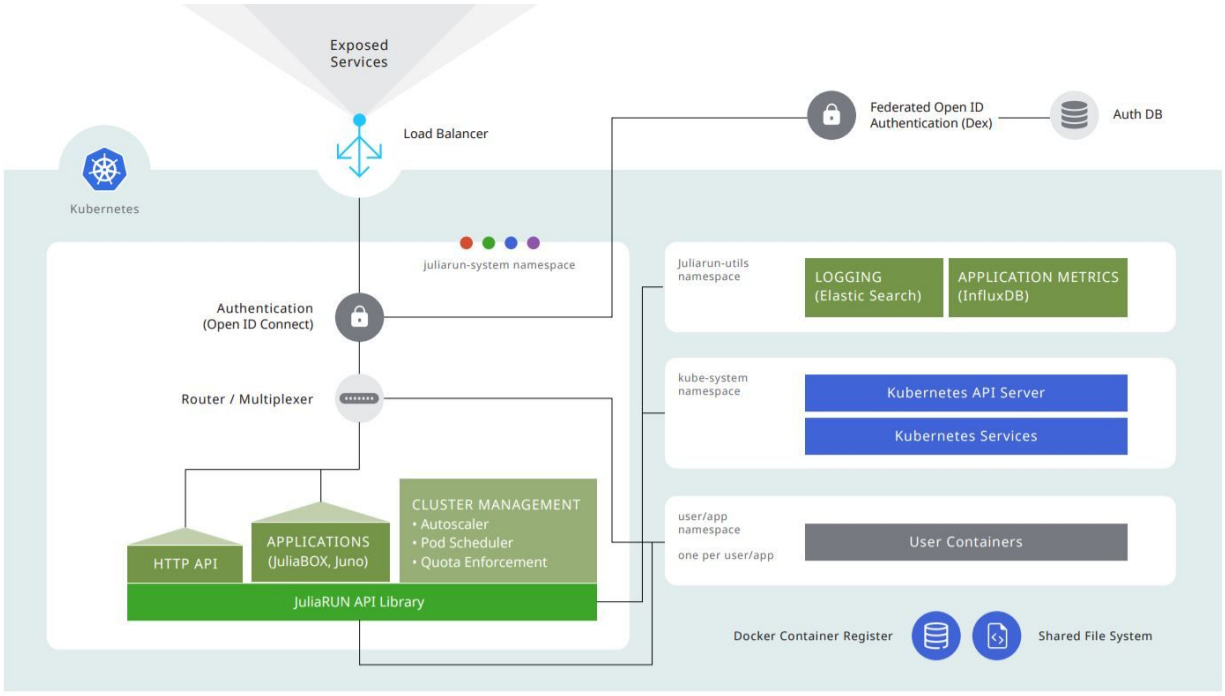
```
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0 (2016-09-19 18:14 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-pc-linux-gnu

julia> ENV["JRUN_CONFIG"] = "config.json"
"config.json"

julia> using JuliaRun
julia> ctx = init()
09-Feb 10:56:48:INFO:root:Compute cluster: default at https://kubernetes:6443
09-Feb 10:56:49:INFO:root:Auth type: token
KuberCluster: Kubernetes namespace default at https://kubernetes:6443

julia> status(ctx)
true

shell> cat /mnt/juliarun/helloworld/helloworld.jl
println("hello, world!")

julia> job = JuliaBatch("helloworld", "/mnt/juliarun/helloworld/helloworld.jl", "juliarun";
                        pkg_bundle="pkgdefault")
JuliaRun.JuliaBatch("helloworld",JuliaRun.EnvSpec("julia","pkgdefault","juliarun",nothing,noth
ing),JuliaRun.ProcSpec("100m","512Mi","/mnt/juliarun/helloworld/helloworld.jl","/mnt/juliarun/
master.sh"))

julia> submit(ctx, job)
09-Feb 10:59:52:INFO:root:starting helloworld...

julia> status(ctx, job)
09-Feb 11:00:29:INFO:root:checking helloworld status...
09-Feb 11:00:30:INFO:root:succeeded: true
(true,0,0,true,false)

shell> cat /mnt/juliarun/helloworld/log/*
hello, world!

julia> delete!(ctx, job)
09-Feb 11:01:24:INFO:root:checking helloworld status...
09-Feb 11:01:24:INFO:root:succeeded: true
09-Feb 11:01:24:INFO:root:deleting helloworld ...
09-Feb 11:01:25:INFO:root:deleting Kuber.Kubernetes.V1Pod helloworld-h9d72 ...
true
```

Hello, World in Parallel

```
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0 (2016-09-19 18:14 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-pc-linux-gnu

julia> ENV["JRUN_CONFIG"] = "config.json";
julia> using JuliaRun
julia> ctx = init();
09-Feb 11:02:35:INFO:root:Compute cluster: default at https://kubernetes:6443
09-Feb 11:02:35:INFO:root:Auth type: token

shell> cat /mnt/juliarun/helloparallel/helloparallel.jl
using JuliaRun

function darts_in_circle(N)
    n = 0
    for i in 1:N
        if rand()^2 + rand()^2 < 1
            n += 1
        end
    end
    n
end

function estimate_pi(N, loops)
    n = sum(pmap((x)->darts_in_circle(N), 1:loops))
    4 * n / (loops * N)
end

function wait_for_workers(min_workers)
    init_parallel(; topology=:master_slave, mode=:master)

    while nworkers() < min_workers
        sleep(5)
    end
end

function master(min_workers, N=10^8, loops=100)
    println("running...")
    wait_for_workers(min_workers)

    est_pi = estimate_pi(N, loops)
```

```
    println("PI (estimated) = ", est_pi)
end

worker() = init_parallel(; topology=:master_slave, mode=:worker)

shell> cat /mnt/juliarun/helloparallel/master.jl
include("helloparallel.jl")
master(2)


shell> cat /mnt/juliarun/helloparallel/worker.jl
include("helloparallel.jl")
worker()

julia> job = JuliaParBatch("helloparallel", "/mnt/juliarun/helloparallel/master.jl",
"juliarun"; pkg_bundle="pkgdefault",
worker_start_script="/mnt/juliarun/helloparallel/worker.jl", nworkers=2);

julia> submit(ctx, job)
09-Feb 11:07:35:INFO:root:starting helloparallel...
09-Feb 11:07:37:INFO:root:waiting for master to start...
09-Feb 11:07:44:INFO:root:starting workers...

julia> status(ctx, job)
09-Feb 11:07:49:INFO:root:checking helloparallel status...
09-Feb 11:07:49:INFO:root:succeeded: false (0/2 workers)
(false,0,2,true,false)

julia> wait(ctx, job)
09-Feb 11:09:08:INFO:root:checking helloparallel status...
09-Feb 11:09:08:INFO:root:succeeded: false (0/2 workers)
09-Feb 11:09:18:INFO:root:checking helloparallel status...
09-Feb 11:09:18:INFO:root:succeeded: false (0/2 workers)
09-Feb 11:09:28:INFO:root:checking helloparallel status...
09-Feb 11:09:29:INFO:root:succeeded: true (2/2 workers)

shell> cat /mnt/juliarun/helloparallel/log/master.stdout
running...
PI (estimated) = 3.1416019364

julia> delete!(ctx, job)
09-Feb 11:11:15:INFO:root:checking helloparallel status...
09-Feb 11:11:15:INFO:root:succeeded: true (2/2 workers)
09-Feb 11:11:15:INFO:root:deleting helloparallel ...
true
```

How JuliaRun works



JuliaRun is a modular architecture comprising of:

1. A JuliaRun server that listens for incoming job requests
2. A Kubernetes based cluster manager, which can be configured for high availability and scalability. Other cluster managers can be used as well, if necessary.
3. A Docker registry that serves docker container images for different job types
4. A shared filesystem for common state
5. Automatic restart of failed jobs

API Reference

Initialization & Authentication

➢ init()

Initialize JuliaRun for the current Julia process and get a context to use with subsequent APIs. The context holds the cluster location, namespace (a cluster partition/isolation) and credentials to authenticate to the cluster with. Authorization settings on the cluster determine resource accessibility and quotas.

Configuration location (a file for now) is picked up from an environment variable. A default authentication context and cluster namespace can optionally be configured in this file. That is enough in a single user scenario and it is not necessary to “authenticate” the context further.

```
using JuliaRun
ENV["JRUN_CONFIG"] = "config.json"
ctx = init()
```

➢ authenticate!(ctx, auth_cfg::Dict{String,Any}, ns::String, server::String)

Set the server location and authentication credentials. Returns the updated context. Overrides any default authentication set from configuration file.

Required parameters:

- auth_cfg: authentication configuration as a JSON type, as mentioned in the configuration section

Optional parameters:

- ns: namespace to authenticate to (default: unchanged)
- server: server address to connect to (default: unchanged)

➢ authenticate(ctx, auth_cfg::Dict{String,Any}, ns::String)

Similar to the above API, but returns a new instance of the context.

Job Submission

JuliaRun supports various different job types:

1. Tasks
    a. Tasks are one time jobs
    b. A Julia program to be run, accompanied by data
    c. Pre-decided list of resources required; the number and type of compute nodes and resources required (compute, memory, storage) on each of them
    d. Queued until resources are available
2. Services
    a. Services are constantly running
    b. Scaled based on activity / thresholds of input channels

Jobs (or a unit/process in case of a parallel job) are restarted if they terminate with a failure status.

➢ submit(ctx, job::JRunJob)

Submit a job definition to execute on the cluster.
Typical job types are bundled with JuliaRun and described in sections below.
New job types can be defined in user code too.

The submitting Julia process can exit after a job submission if required. It can resume later with a new JuliaRun context and continue further operations with the same job.

➢ endpoint(ctx, job::JRunJob)

Get the endpoint exposed by the job/service.

Certain jobs (services, e.g. Notebook, Webserver, MessageQ) can expose network ports for users or other jobs to interact.

Returns a tuple of:

- Vector{String} - IP addresses / hostnames exposed
- Dict{String,Int} - Port name and port numbers exposed

➢ **endpoint_proxy(ctx, job::JRunJob, port_name::String="")**

Get the proxied endpoint of a job/service exposed by the cluster. This is valid only for HTTP endpoints. A proxied endpoint may provide additional functionalities (e.g. security) over an internal endpoint exposed by a job/service.

Returns a string pointing to the endpoint URL.

Job Types

➢ **JuliaBatch(name, start_script, run_volume; optional_args...)**

A Julia Batch Job.

Runs in a single allocation/container restricted to the specified amount of CPUs and memory. The Julia process started in the batch job is allowed to start other worker processes or threads, all within the specified limits. The upper bound of CPUs/memory is determined by the largest cluster node.

Scenarios:

- non-parallel
- multi-threaded parallelism
- multi-process shared memory parallelism

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **start_script**: a Julia script to start the master node with
- **run_volume**: a persistent volume to use as the master volume

Optional parameters:

- **pkg_bundle**: a package bundle to set at LOAD_PATH (none by default)
- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **ports**: network ports to expose as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: nothing)
- **external**: whether network ports should be exposed to external network (default: false)
- **image**: the docker image to run (default: julia)
- **cpu**: CPU share to allocate (default: 0.1 of a core)
- **gpu**: GPUs to allocate (default: 0)
- **memory**: RAM to allocate (default: 512 MiB)
- **shell**: the script to use for startup shell (default: pre-configured master.sh)

➢ JuliaParBatch(name, start_script, run_volume; optional_args…)

A Julia Master-Slave Parallel Batch Job.

A designated master process and an array of worker processes, each of which runs in a separate allocation/container. All worker processes have a uniform resource allocation. The master process can have a different allocation.

The job as a whole can use all available CPU/memory in a cluster (within practical limitations imposed networking and such). The upper bound for each master/worker process is determined by the largest node in the cluster.

The master and worker processes are initialized for Julia parallel constructs. The number of workers can be checked and manipulated with the `scale`/`scale!` APIs. The number of worker processes actually running depends on the available cluster resources and may be different from the requested scale. The job is deemed running when the master process starts, and JuliaRun attempts to match number of worker processes to the requested scale continuously. The master process may check the number of actual workers with the `nworkers` API and wait for a minimum number of workers if needed by it.

Scenario:

- multi-process distributed memory parallelism

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **start_script**: a Julia script to start the master node with
- **run_volume**: a persistent volume to use as the master volume

Optional parameters:

- **pkg_bundle**: a package bundle to set at LOAD_PATH (none by default)
- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **ports**: network ports to expose from the master process as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: nothing)
- **external**: whether network ports should be exposed to external network (default: false)
- **image**: the docker image to run (default: julia)
- **cpu**: CPU share to allocate to master process (default: 0.1 of a core)

- **gpu**: GPUs to allocate (default: 0)
- **memory**: RAM to allocate to master process (default: 512 MiB)
- **shell**: the script to use for master shell (default: master.sh)
- **nworkers**: number of worker processes to start (0 by default, can be scaled later)
- **worker_cpu**: CPU share to allocate to a worker process (default: 0.1 of a core)
- **worker_gpu**: GPUs to allocate to a worker process (default: 0)
- **worker_memory**: RAM to allocate to a worker process (default: 512 MiB)
- **worker_shell**: the script to use for a worker shell (default: master.sh)
- **worker_start_script**: a Julia script to start the worker node with (default: worker.sh)

➢ JuliaParBatchWorkers(name, start_script, run_volume; optional_args...)

A Julia Embarrassingly Parallel Batch Job.

An array of batch jobs that can be scaled as a single unit. Each unit in the job array can be likened to a JuliaBatch job. Since there is no interlinking between the units, this can be scaled faster and to much higher levels.

As a whole, the job can use all available CPU/memory in the cluster, but the upper bound for each worker process is determined by the largest node in the cluster.

Scenarios:

- work queues
- distributed memory parallelism

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **start_script**: a Julia script to start each node with
- **run_volume**: a persistent volume to use as the work volume

Optional parameters:

- **pkg_bundle**: a package bundle to set at LOAD_PATH (none by default)
- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **ports**: network ports to expose as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: nothing)
- **external**: whether network ports should be exposed to external network (default: false)
- **image**: the docker image to run (default: julia)
- **cpu**: CPU share to allocate to each process (default: 0.1 of a core)

- **gpu**: GPUs to allocate to each process (default: 0)
- **memory**: RAM to allocate to each process (default: 512 MiB)
- **shell**: the script to use for process shell (default: worker.sh)
- **nworkers**: number of worker processes to start (0 by default, can be scaled later)

➢ Notebook(name, run_volume; optional_args...)

A Julia Interactive Notebook.

The notebook is accessible outside the cluster and can be protected by a password (or an external portal/proxy). The master container runs the Jupyter notebook process and the Julia kernels.

The Julia process running as the kernel is similar to a JuliaBatch.
Workers can be provisioned (optionally) as:

- slaves to a master Julia process (a Jupyter kernel), similar to JuliaParBatch
- a job array, similar to JuliaParBatchWorkers

Scenarios:

- interactive/exploratory tasks
- shell access (through Jupyter shell)
- file transfer (through Jupyter file manager)

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **run_volume**: a persistent volume to use as the master volume

Optional parameters:

- **pkg_bundle**: a package bundle to set at LOAD_PATH (none by default)
- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **image**: the docker image to run (default: julia)
- **cpu**: CPU share to allocate to master process (default: 0.1 of a core)
- **gpu**: GPUs to allocate to master process (default: 0)
- **memory**: RAM to allocate to master process (default: 512 MiB)
- **shell**: the script to use for master shell (default: notebook.sh)
- **worker_cpu**: CPU share to allocate to a worker process (default: 0.1 of a core)
- **worker_gpu**: GPUs to allocate to a worker process (default: 0)
- **worker_memory**: RAM to allocate to a worker process (default: 512 MiB)
- **worker_shell**: the script to use for a worker shell (default: master.sh)

- **worker_start_script**: a Julia script to start the worker node with (default: worker.sh)
- **passwd**: password to protect the notebook with (default: no password)
- **ports**: network ports to expose as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: Dict("nb"=>(8888,8888)))
- **external**: whether network ports should be exposed to external network (default: false)

➢ PkgBuilder(name, builder_script, run_volume, pkg_bundle; optional_args...)

Build / update a Julia package bundle.
It mounts the volume representing the package folder of the bundle in read-write mode, and launches a script to build them. It is similar to a JuilaBatch job.

Package bundles are a way of making Julia packages accessible to JuliaRun jobs. They are just folders with Julia packages, pre-compiled and along with all their dependencies. Package bundles can be built and tested separately and attached to multiple images, providing a way to dissociate their maintenance.

Attaching a package bundle to a JuliaRun job sets the appropriate environment variables (`LOAD_PATH` and such) so that Julia code can start using them seamlessly.

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **builder_script**: a Julia script that builds packages
- **run_volume**: a persistent volume to use as the master volume
- **pkg_bundle**: the package bundle to build

Optional parameters:

- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **image**: the docker image to run (default: julia)
- **cpu**: CPU share to allocate (default: 2 cores)
- **gpu**: GPUs to allocate (default: 0)
- **memory**: RAM to allocate (default: 8 GiB)
- **shell**: the script to use for startup shell (default: master.sh)

➢ Webserver(name, cfg_file::String, run_volume; optional_args...)

Webserver that can be used to serve API/UI/files for jobs.

Scenarios:

- accept inputs (file uploads, API calls, ...)
- serve output/log files created by jobs
- simple user interfaces

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **cfg_file**: configuration file for the web server
- **run_volume**: a persistent volume to use as the master volume

Optional parameters:

- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **image**: the docker image to run (default: nginx)
- **cpu**: CPU share to allocate (default: 0.1 cores)
- **memory**: RAM to allocate (default: 512 MiB)
- **envvars**: additional environment variables (none by default)
- **ports**: network ports to expose as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: Dict("http"=>(80,80), "https"=>(443,443)))
- **external**: whether network ports should be exposed to external network (default: false)
- **nworkers**: number of webserver containers to start

➢ MessageQ(name, cfg_file::String, run_volume; optional_args...)

Message Queue that can be used for communication between processes within or across jobs. Enables complex routing/broadcast of messages.

Scenarios:
- inter process communication within / across jobs
- task queue, used by job arrays

Required parameters:

- **name**: name of the job (must be unique in the system, can be random)
- **cfg_file**: configuration file for the queue server
- **run_volume**: a persistent volume to use as the master volume

Optional parameters:

- **additional_volumes**: more persistent volumes to attach at /mnt/<volume name> (none by default)
- **image**: the docker image to run (default: rabbitmq:3)
- **cpu**: CPU share to allocate (default: 1 core)
- **memory**: RAM to allocate (default: 4 GiB)
- **envvars**: additional environment variables (none by default)
- **ports**: network ports to expose as a Dict{String,Tuple{Int,Int}} of name to container port and service port map (default: Dict("amqptls"=>(5671,5671), "amqp"=>(5672,5672), "mgmttls"=>(15672,15671), "mgmt"=>(15672,15672)))
- **external**: whether network ports should be exposed to external network (default: false)

Monitoring

➢ status(ctx; optional_args...)

Get the status of the cluster (Bool, up/down)

Additional keyword parameters:

- **verbose**: output node and capacity details as well

➢ status(ctx, job::JRunJob)

Get the current status of a job.

Returns an object of type JobStatus with the following fields:

- created::Bool: whether the job has been scheduled
- succeeded::Bool: whether the job completed
- nworkers_succeeded::Int: for a parallel job, number of workers that completed successfully
- parallelism::Int: for a parallel job, number of workers requested
- npending::Int: number of processes pending creation (waiting for cluster resources)

➢ scale(ctx, job::JRunJob)

Get the current scale of the job. Scale of a job refers to the number of parallel ‘worker’ processes. A job with a single master process has a scale of 0.

Returns a tuple of:

- ➢ Int: number of workers running
- ➢ Int: number of workers requested

➢ list(ctx)

List all visible JuliaRun jobs in the cluster.

Returns a Vector{JRunJob}.

➢ tail(ctx; optional_args...)

Tail logs. All logs from the namespace pointed to by the context are tailed by default.

Logs can be filtered by specifying keyword parameters:

- **job**: a JRunJob instance
- **stream**: "stdout" or "stderr" (both by default)

Additional keyword parameters:

- **output**: an IO stream or Channel to output the log entries on
- **count**: number of log entries to fetch (default as determined by logging system)
- **follow**: whether to keep tailing (stops when output is closed or job is finished)

Scaling

➢ init_parallel(;kwargs...)

Initialize Julia processes to use Julia parallel constructs.
Parameters:

- **topology**: Julia parallel topology to use (:master_slave by default)
- **mode**: Whether to initialize as master (default) or worker (:master or :worker)
- **stdout_to_master**: whether to redirect stdout/stderr of workers to master (false by default)
- **connect_retries**: number of times workers retry connecting with master, if they come up before master is ready (10 by default)

➢ wait_for_workers(min_workers)

Wait till the specified number of workers join the master.

➢ scale!(ctx, job::JRunJob, parallelism::Integer)

Request to scale the job up or down to the level of parallelism requested.

When scaling up, whether the target level is achieved or not depends on available cluster resources. A scaling request can remain in a partially satisfied condition till required resources become available (either by resizing the cluster or due to other jobs releasing resources).

➢ scale!(ctx, job::JRunJob, steps::Range)

Scaling can be done in steps by providing a step range instead of a hard target. JuliaRun would then wait for the previous scale! request to be satisfied at each step.

This is useful to avoid flooding common resources while scaling (e.g. connect storm).

➢ Scaling Processes with JuliaRun.Scaler

The Scaler sub module in JuliaRun provides a framework and a default implementation to autoscale Jobs.

Using JuliaRun.Scaler, JuliaRun can be setup to periodically monitors a metric to determine whether and how much to scale a Job up or down.

Metrics are implementations of `ScalerMetric{T}`. They monitor a resource/attribute published by a cluster component or a running Job, transform it appropriately and provide a value of type T when queried. A `ScalerMetric` can can monitor multiple resources/attributes as well and have complex rules based on them.

JuliaRun bundles a few implementations of `ScalerMetric`, more can be added through external code. The ones that are bundled now are:

- **JobMetric:** Metric collected by JuliaRun or published by the job. E.g. CPU/memory.
- **WorkQueueMetric**: measures a work queue (message queue) size
- **JobScaleMetric**: derives value from a given metric and the current job scale using a Julia function
- **Average**: averages a metric over a specified window size
- **Bounded**: keeps a metric bounded between a minimum and maximum
- **Threshold**: applies upper and lower threshold values on a metric
- **Combine**: combine multiple metrics using a Julia function

Scalers are implementations of abstract `JRunScaler`. JuliaRun bundles one implementation, more can be added through external code. The bundled implementation is:

- **JobScaler <: JRunScaler**: scales a JuliaRun job

Scaler terminates when the target resource or metric terminates. A scaler itself is a JuliaRun job, and thus be resilient to failures. Scalers can be created and deleted independently of the jobs they scale. However deleting a job also cleans up any active scalers set up for it.

Example usage:

```
# scale job based on cpu, by 1 worker each time
# scale up when usage goes beyond 80%
# scale down when usage drops below 60%
scaler = JobScaler(job,
         ctx,
         JobScaleMetric(Threshold(JobMetric("cpu"), (60,80), 1)))
submit(ctx, scaler)
```

JuliaRun autoscaler can scale based on CPU, memory or any custom metric published by the job. A Julia function can be provided when complex logic is needed. Kubernetes Pod autoscaler is also supported, and works only for CPU based scaling.

```
# scale job between 1 and 5 when cpu usage crosses 70%
scaler = JuliaRun.KuberClusters.KuberJobScaler(job, 70, 1, 5)
submit(cm, scaler)
```

➢ Scaling Cluster Nodes

JuliaRun also includes a cluster autoscaler that can add or removed nodes based on load. The autoscaler is made available bundled as a docker container which can be scheduled as a JuliaRun service. To configure the cluster autoscaler look at the configuration section. To schedule (customize to match deployment specifics):

```
ENV["JRUN_CONFIG"] = "config.json"
using JuliaRun
ctx = init()
namespace!(ctx, get(ctx, :Namespace, "kube-system"))
job = JuliaBatch("autoscaler", "/home/jrun/monitor.jl", "juliarun";
                 cpu="1", memory="2Gi",
                 image="juliarunautoscaler",
                 shell="/home/jrun/internal_master.sh")
submit(ctx, job)
```

Termination

➢ delete!(ctx, job::JRunJob; force::Bool=false)

Removes the job entry from the queue.
To remove an incomplete job, specify force to be true.
Job entries are kept in the queue even after completion and can be queried for status.

➢ wait(ctx, job::JRunJob)

Blocking wait for a job to complete.

Metrics

JuliaRun provides some APIs to monitor performance metrics and resource usage. See the "Configuration" section also for related settings. CPU and memory usage is already logged. Applications can publish custom metrics.

➢ setup_metrics(ctx)

Setup metrics collection and accounting on the cluster, essentially setting up few default aggregation rules and data retention rules in the data store. This only does a default minimal setup, good enough for a simple setup. For larger production clusters, external manual setup may be necessary.

➢ cleanup_metrics(ctx)

Cleans up all the settings done through `setup_metrics`. Also deletes any aggregated data that were stored in them. For larger production clusters, this would likely need to be done externally.

➢ record_metrics(ctx, resource, metric; at_time=nothing, tags=nothing)

Record a metric against the resource, tag it with `tags`.

Parameters:

- **resource**: name of the resource being monitored (String)

- **metric**: value of the metric (Float64)
- **at_time**: nanosecond time to record this at (current time taken if not provided)
- **tags**: A Dict{String,String} of tag names and values to set with this record. Tags are needed for aggregation and searching. By default JuliaRun uses `container_name` and `namespace_name` tags.

➢ record_metrics(ctx, metrics; at_time=nothing, tags=nothing)

Record more than one metrics against the resource.

Parameters:

- **metrics**: name-value metric pairs (Dict{String,Float64})

➢ get_rusage(ctx, start_time=0, end_time=0, resource=nothing)

Get resource usage for the current context aggregated over the specified period.

Parameters:

- **start_time:** time (inclusive) to consider from, either in absolute nanoseconds or relative (default: “now()”)
- **end_time**: time (inclusive) to consider till, either in absolute nanoseconds or relative (default: “now() - 1h”)
- **resource**: single or an array of resource names (default: ["cpu","mem"])

➢ get_metrics(ctx, start_time=0, end_time=0, metrics=[])

Get aggregated metrics per container over the specified period.

Parameters:

- **start_time:** time (inclusive) to consider from, either in absolute nanoseconds or relative (default: “now()”)
- **end_time**: time (inclusive) to consider till, either in absolute nanoseconds or relative (default: “now() - 10m”)
- **metrics**: list of resource names and type of aggregation to do on them (mean cpu and memory by default, [(“cpu”,”mean”),(“mem”,”mean”)])

➢ show_metrics(ctx, start_time, end_time, metrics; times=1, delay_secs=5)

Similar to `get_metrics`, but displays the results in tabular format. Useful for continuously monitoring activity.

Parameters:

- **start_time:** time (inclusive) to consider from, either in absolute nanoseconds or relative (default: “now()”)
- **end_time**: time (inclusive) to consider till, either in absolute nanoseconds or relative (default: “now() - 10m”)
- **metrics**: list of resource names and type of aggregation to do on them (mean cpu and memory by default)

Optional parameters:

- **times:** number of times to repeat, specifying ‘0’ makes it go endlessly
- **delay_secs**: delay between repeats

HTTP API Reference

JuliaRun can be started as a service, whereby it exposes HTTP endpoints that can be used to submit and manage jobs. Only a subset of JuliaRun APIs are available via HTTP.

To start JuliaRun as a service:

```
#---------------------------
ENV["JRUN_CONFIG"] = "config.json"
using JuliaRun
using JuliaRun.Scaler

ctx = init()

namespace!(ctx, get(ctx, :Namespace, "juliarun"))

job = JuliaParBatchWorkers("juliarunremote", "/home/jrun/remote_inproc.jl", "juliarun";
              nworkers=1, cpu="1", memory="2Gi", image="juliarunremote",
              shell="/home/jrun/internal_master.sh", ports=Dict("http"=>(8888,80)))
submit(ctx, job)

const CHECK_INTERVAL = 60*5
cpu_or_mem = Combine([JobMetric("cpu"), JobMetric("memory")], maximum)
metric = Bounded(JobScaleMetric(Threshold(cpu_or_mem, (60.0,80.0), 1)), 1:5)
scaler = JobScaler(job, ctx, metric, CHECK_INTERVAL)
submit(ctx, scaler)
#---------------------------
```

API endpoints are served by JuliaWebAPI using its JSON protocol. They can be called with the below example code, or its equivalent:

```
#--------------------------
using Requests
using JSON

function result(hresp)
    (hresp.status == 200) || error("HTTP error: ", hresp.status, ",", String(hresp.data))
    resp = JSON.parse(readall(hresp))
    (resp["code"] == 0) || error("API error: ", resp["code"], ", ", resp["data"])
    resp["data"]
end

result(get("http://localhost:8888/getSystemStatus"))
#--------------------------
```

A JuliaRunClient package is also available to make use of the HTTP API from Julia code. The package can be installed with:

```
Pkg.clone("https://github.com/JuliaComputing/JuliaRunClient.jl.git")
```

Authentication:

API Token:

API calls can be authenticated by setting up an authentication filter to validate an API token.

Open ID Connect / Kubernetes Service Accounts:

All API endpoints accept optional authentication parameters:

- **jruntok**: A base64 encoded token to authenticate with. Can be an OpenID connect token or a Kubernetes service account token. To use OpenID connect, Kubernetes must be configured with an OpenID service provider.
- **jrunns**: The cluster namespace to authenticate with.

API endpoints:

➢ /reconfigure

Reloads configuration file and applies any changes.
Returns:

- Bool: true/false indicating success/failure

➢ /getSystemStatus

Verifies if JuliaRun is running and is connected to a compute cluster.
Returns

- Bool: true/false indicating success/failure

➢ /getJobStatus/<JobType>/?name=<JobName>

Fetch current status of a Job.
JobType: One of the job types listed in the manual
Parameters:

- name: job name to get the status of

Returns tuple/array with:

- Bool: whether the job completed
- Int: for a parallel job, number of workers that completed successfully
- Int: for a parallel job, number of workers started
- Bool: whether the job has been created (vs. scheduled)
- Bool: whether this is a notebook (legacy, likely to be removed in future)

➢ /getJobScale/<JobType>/?name=<JobName>

Get the current scale of the job.
JobType: One of the job types listed in the manual or in the section below
Parameters:

- name: job name to get the scale of

Returns tuple/array with:

- Int: number of workers running
- Int: number of workers requested

➢ /setJobScale/<JobType>/?name=<JobName>&parallelism=<scale>

Request to scale the job up or down to the level of parallelism requested.
JobType: One of the job types listed in the manual or in the section below
Parameters:

- name: job name to set the scale of
- parallelism: number of workers to scale to

Returns:

- Bool: true/false indicating success/failure

➢ /getJobEndpoint/<JobType>/?name=<JobName>

Get the endpoint exposed by the job/service.
JobType: One of the job types listed in the manual
Parameters:

- name: job name to get the endpoint of

Returns a tuple/array of endpoints as URLs or IP and ports

➢ /getAllJobInfo

Dict of all status information of all current jobs.
Caution: this API is not built for scale yet.
Returns a dict of the form:

```
{"jobname": { "type": "JobType", "status": [], "scale": [],
"endpoint": [] }...}
```

➢ /listJobs

List all submitted jobs.
Returns a dictionary similar to the one returned in `/getAllJobInfo`, but with only the `type` attribute populated for each job.

➢ /tailJob/<JobType>/?name=<JobName>&stream=stdout&count=0

Tail logs from the job. Returns a string of log entries separated by new line.
JobType: One of the job types listed in the manual
Parameters:

- name: job name to delete
- stream: the stream to read from (stdout/stdin), all streams are read if not specified.
- count: number of log entries to return (50 by default)

➢ /deleteJob/<JobType>/?name=<JobName>&force=<true/false>

Removes the job entry from the queue.
JobType: One of the job types listed in the manual or in the section below

Parameters:

- name: job name to delete
- force: whether to remove an incomplete job (optional, default: false)

Returns:

- Bool: true/false indicating success/failure

➢ /submitJob/<JobType>/?name=<JobName>&<job specific parameters>

Submit a job definition to execute on the cluster.
JobType: One of the job types listed in the manual or in the section below
Parameters:

- name: job name to delete
- job specific parameters, with names as documented for the JobType constructor

Returns nothing.

The optional “ports” parameter, which is a Dict{String,Tuple{Int,Int}} in the Julia API, can be sent as a comma separated list of “String:Int,Int”.
E.g.: “http:80:80,https:443:443”

If the port name is omitted, it defaults to port suffixed with the container port number.
E.g.: “80:80,443:443” implies “port80:80:80,port443:443:443”

If the container port is omitted, it defaults to the service port number:
E.g.: “80,443” implies “port80:80:80,port443:443:443”
E.g.: “http:80,https:443” implies “http:80:80,https:443:443”

Configuration

JuliaRun is configured from a JSON file. Configuration can be re-loaded without affecting running jobs. Below is a sample configuration file with inline comments

Logging

Logging configuration for JuliaRun.

```
"log": {
    "level": "INFO",
    "filename": "juliarun.log"
}
```

Cluster Type

Configures the modules that implements the job management functions required for a certain cluster backend:

- The compute backend. E.g.: KuberCluster for a Kubernetes backend.
- The log management backend.
    - KuberLogger: default kubernetes pod level log management
    - ESLogger: Elasticsearch cluster level logger
- Mechanism of exposing external services in the cluster.
    - LoadBalancer: on public clouds (GCE/AWS)
    - NodePort: local single node machines, or a cloud setup with no load balancer support.

```
"clustertype": "JuliaRun.KuberClusters.KuberCluster",
"externalsvc": "LoadBalancer",
"loggertype": "JuliaRun.KuberClusters.KuberLogger",
```

Compute Resources

Compute resources configuration, specific to the cluster backend. For a Kubernetes cluster one can specify the compute node (api server) address. Namespace specifies the Kubernetes namespace to use.

```
"compute": {
    // uri points either to
    // - a proxy (kubectl) with auth type "none"
    // - to kubernetes api server with auth settings populated
    "uri": "https://kubernetes:6443/",
    "namespace": "default",
    "auth": {
        // type can be:
        // "none": auth through proxy
        // "cert": client cert (requires client-certificate-data
        //         and client-key-data)
        // "token": token auth (requires openid connect/service token)
        "type": "token",
        // ca certificate to authenticate the server with
        "certificate-authority-data": "base64 encoded cacert",
        "cert-verify": "true",
        "client-certificate-data": "base64 encoded client cert",
        "client-key-data": "base64 encoded client key",
        "token": "base64 encoded token"
```

```
    }
},
```

Storage Resources

All storage units preconfigured. They are referenced only by volume name in jobs. Therefore, volume names must be unique across storage types.

```
"storage": {
    // one or more glusterfs clusters
    "glusterfs": [
        {
            "name": "glusterfs",
            // ip addresses of the glusterfs nodes
            "ip": ["10.128.0.3", "10.128.0.4", "10.128.0.5"],
            // volumes available to mount from the glusterfs cluster
            "mount": [
                {"volume": "datavol1", "name": "juliarun",
                 "capacity": "1Ti", "readonly": false},
                {"volume": "datavol2", "name": "data",
                 "capacity": "4Ti", "readonly": false}
            ]
        }
    ],
    // nfs or amazon efs volumes
    "nfs": [
        {
          "server": "fs-ab9d5e02.efs.us-west-2.amazonaws.com",
          "path": "/juliarun",
          "name": "juliarun",
          "readonly": false,
          "capacity": "1Ti"
        }
    ],
    // azure file share
    // credential refers to a secret created in relevant
    //   namespaces with the azure storage account name and key
    "azurefs": [
        {
            "credential": "azurefs-secret",
            "share": "juliarun",
            "name": "juliarun",
            "readonly": false
        }
    ],
    // azure vhd disks
```

```
"azuredisk": [
    {
        "name": "sampledataro",
        "disk": "disk.vhd",
        "uri": "https://someaccount.blob.microsoft.net/vhds/disk.vhd",
        "readonly": false
    }
],
// git repositories to make available as volumes
// volumes are read-write but modifications are not checked in
"gitrepo": [
    {
        "name": "examplerepo",
        "repository": "https://github.com/JuliaLang/Example.jl"
    }
],

// google cloud disks
"gce": [
    { "name": "sampledataro", "disk": "sample-data-disk",
      "readonly": true },
    { "name": "sampledatarw", "disk": "sample-data-disk",
      "readonly": false },
    { "name": "pkgdefault", "disk": "sample-pkgbundle-disk",
      "readonly": true },
    { "name": "pkgdefaultrw", "disk": "sample-pkgbundle-disk",
      "readonly": false }
],

// amazon ebs volumes
// ebs volume can be used only with a single JuliaBatch job
// because of inherent restrictions of the type of volume
"ebs": [
    { "name": "sampledataro", "volume_id": "vol-1a123456",
        "fs_type": "ext4", "readonly": true },
    { "name": "sampledatarw", "volume_id": "vol-1a123456",
        "fs_type": "ext4", "readonly": false }
],

// scratch disks, temporary non-persistent storage,
// sourced from host machine
"scratch": [
    { "name": "empty", "readonly": false }
],
// host folders mounted as volumes (useful only in single node
// setups, development setups)
"hostpath": [
    { "name": "testhostpath", "path": "/mnt/testhostpath",
```

```
            "readonly": false }
        ]
    }
```

Images

Docker images (and the repository to pull them from) mapped to abstract image names used in job templates. Multiple repositories may be used in the configuration, by qualifying images with the repository name.

```
    "images": {
        "julia": {
            "image": "gcr.io/jrun/juliarun/julialatest:v0.0.4",
            "executable": "/opt/julia-0.5/bin/julia",
            "juliaver": "v0.5"
        },
        "julia-0.5": {
            "image": "gcr.io/jrun/juliarun/julialatest:v0.0.4",
            "executable": "/opt/julia-0.5/bin/julia",
            "juliaver": "v0.5"
        },
        "julia-0.4": {
            "image": "gcr.io/jrun/juliarun/julialatest:v0.0.4",
            "executable": "/opt/julia-0.4/bin/julia",
            "juliaver": "v0.4"
        },
        "nginx": {
            "image": "gcr.io/jrun/nginx:latest",
            "executable": "/usr/sbin/nginx"
        },
        "rabbitmq": {
            "image": "gcr.io/jrun/rabbitmq:3-management"
        }
    }
```

Packages

Packages are special volumes that contain Julia packages. For each package listed, there must be a readonly and a readwrite volume (name suffixed with 'rw') configured.

```
    "packages": ["pkgdefault"],
```

Metrics

On a Kubernetes cluster, Heapster with InfluxDB is required.

```
// accountingtype specifies the type of resource accounting implementation
// Available options:
// - JuliaRun.KuberClusters.NoAccounting:
//   no accounting information gathered (default)
// - JuliaRun.KuberClusters.InfluxDBAccounting:
//   heapster+influxdb monitoring based accounting
"accountingtype": "JuliaRun.KuberClusters.NoAccounting",

"accounting": {
    // window to calculate per container averages of cpu and memory
    "average": "10m",
    // window to aggregate averages for resource accounting
    "aggregate": "60m",
    // aggregate/average ratio
    // (used to calculate aggregate value from sum of averages)
    "aggregate_ratio": "6",
    // window to retain data till
    "retention": "480h",
    // window for each data shard (influxdb specific)
    "shard_duration": "48h",
    // replication for replication (influxdb specific)
    "replication": "1",
    // the tag/column name that identifies the group for which data
    // to be aggregated for resource accounting
    "tag_account_name": "namespace_name",
    // the retention policy name to use (influxdb specific)
    "retention_policy": "default",
    // the database to use
    "db": "k8s",
    // how to initialize retention policy (influxdb specific)
    "init_retention_policy": "alter",
    // whether to set the retention policy as default (influxdb specific)
    "default_retention_policy": "true"
}
```

Cluster Autoscaler

```
// scalingtype implements cluster auto scaling
// Available options:
// - JuliaRun.KuberClusters.NoScaling : no scaling is done
// - JuliaRun.KuberClusters.KuberScaler : scales up a cloud (AWS, Azure, GCE)
```

```
//   cluster based on resources reserved while maintaining a configurable
//   headroom
"scalingtype": "JuliaRun.KuberClusters.NoScaling",

// KuberScaler configuration parameters
"kuberscaler": {
    // type of cloud scaler to use
    // Available options:
    // JuliaRun.KuberClusters.AzureScalers.AzureACSEngineScaler
    // JuliaRun.KuberClusters.AWSScalers.AWSScaler
    "cloudscaler": "JuliaRun.KuberClusters.AWSScalers.AWSScaler",
    // resource headroom to maintain at different cluster sizes
    // smaller clusters typically need larger headroom
    "headroom": {
        "nodecount": [3, 5, 10, 20],
        "freepct": [30, 20, 15, 10]
    },
    // minutes between scaling activities
    "cooldown": 5,
    "min": 1,
    "max": 5
}
```