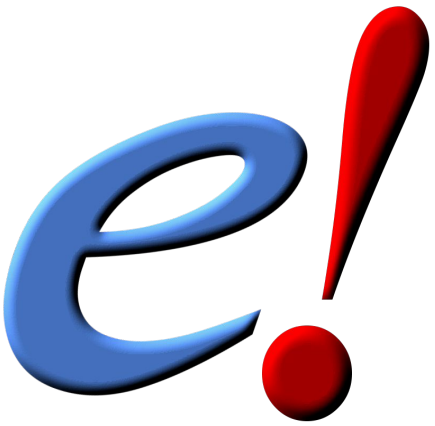
eHive Workshop

part 4: writing your own Runnables

Instructors:
Leo Gordon and Brandon Walts

EMBL-EBI

wellcome trust sanger institute



States of a Job

- A Job is a parameterized Storable instance of a Runnable.
  It is fully represented in the Hive database (job table, job_id, foreign keys, etc)

- A Job goes through the following states:

  - [SEMAPHORED] -- if they are created in pre-blocked state
  - READY -- can be claimed by Workers
  - CLAIMED -- for a short period to ensure no race condition with other Workers
  - [PRE_CLEANUP] -- method -- mostly file/db cleanup after prev. attempt
  - FETCH_INPUT -- method -- checking parameters and database activity
  - RUN -- method -- main functionality, ideally mute
  - WRITE_OUTPUT -- method -- mostly writing into databases, dataflow
  - [POST_CLEANUP] -- method -- mostly memory cleanup
  - DONE -- this is how they all should be
  - [FAILED] -- if exhausted all attempts
  - [PASSED_ON] -- if garbage-collected from a killed Worker

Lifecycle of a Runnable/Job

- Hive Runnables inherit from `'Bio::EnsEMBL::Hive::Process'` (or its descendents). It gives them two things:
  - they get access to Hive API (the visible part of which is parameter management)
  - they acquire a lifecycle() subroutine that calls the following “virtual” methods:
    - param_defaults() # a hash of the lowest level defaults in parameter precedence
    - pre_cleanup() # is only called for retry_counts>0, mainly to clean up files
    - fetch_input()
    - run()
    - write_output()
    - post_cleanup() # mainly to clean up memory after all values of retry_count

standaloneJob.pl

- standaloneJob.pl is a script to run a parameterised Runnable without a database at all:
  - You can pass the param as command line options:

```
standaloneJob.pl Bio::EnsEMBL::Hive::RunnableDB::SystemCmd -cmd 'ls -l'
```

  - Or, supply params as an -input_id hash

```
standaloneJob.pl Bio::EnsEMBL::Hive::RunnableDB::SystemCmd \
    -input_id "{ 'cmd' => 'ls -l' }"
```

Parameter retrieval/storage API

- Jobs do not know where the parameters they are working with come from.
  All they need to know is:
  - How to get a value of a parameter:
    ```
    my $alpha = $self->param('alpha');
    ```
  - How to set it to make available to other parts of Job's lifecycle:
    ```
    $self->param('beta', $beta);
    ```
  - How to require that the given parameter has been passed:
    ```
    my $alpha = $self->param_required('alpha');
    ```
  - How to check whether it is defined:
    ```
    if ($self->param_is_defined( 'gamma' ) ) { … }
    ```
  .
- Parameters that you have stored in $self->param() are not automatically dataflown anywhere, it is your responsibility to trigger Dataflow Events.

Dataflow API: creating events

- A dataflow event has two parameters: a hash of parameters and branch number:

```
$self->dataflow_output_id( { 'alpha' => 1.5, 'gamma' => 5 }, 3 );
```

- The first parameter can also be an arrayref (of hashes):

```
$self->dataflow_output_id( [{'name' => 'Alice'}, {'name' => 'Bob'}], 2);
```

- Feel free to use any number of distinct dataflow branches to create events, they do not have to be all wired. You can create different modes of operation by wiring different branches. A separate branch_number should be allocated for each distinct kind of data.

- Be careful when explicitly dataflowing into branch #1, as this will override the autoflow. You shoud know what you are doing. In particular, multiple events in branch #1 is a bad idea.

- If you do explicitly dataflow into branch #1, make sure this Dataflow Event happens after all Dataflow Events you envisage may constitute a fan with funnel in branch #1.

Error reporting API

- You may leave a non-fatal human-readable message in log_message table:

```
$self->warning( 'I got a strange feeling I am in an infinite loop…' );
```

- Do not mix it with “warn” whose output will go to wherever STDERR of the Job is

- Any fatal message will also be recorded in log_message:

```
die 'all gone wrong';   # just the message
$self->throw();     # with call stack trace (including Hive internal calls)
```

- However the same die/throw/croak/... calls can be used to mark the successful completion of a Job. In this case you have to first unset the incomplete flag.

```
$self->input_job->incomplete( 0 );
die 'all gone right';  # this message is still recorded
```

- Setting transient_error to 0 and then dying will prevent further attempts to retry the Job:

```
$self->input_job->transient_error( 0 );
if( $alpha < 0) { die “alpha parameter cannot be negative”; }
$self->input_job->transient_error( 1 );
```

- You can also instruct the Worker to exit if you believe it has been contaminated:

```
$self->input_job->lethal_for_worker( 1 );
die “There is no point to carry on with this Worker: /tmp is full”;
```

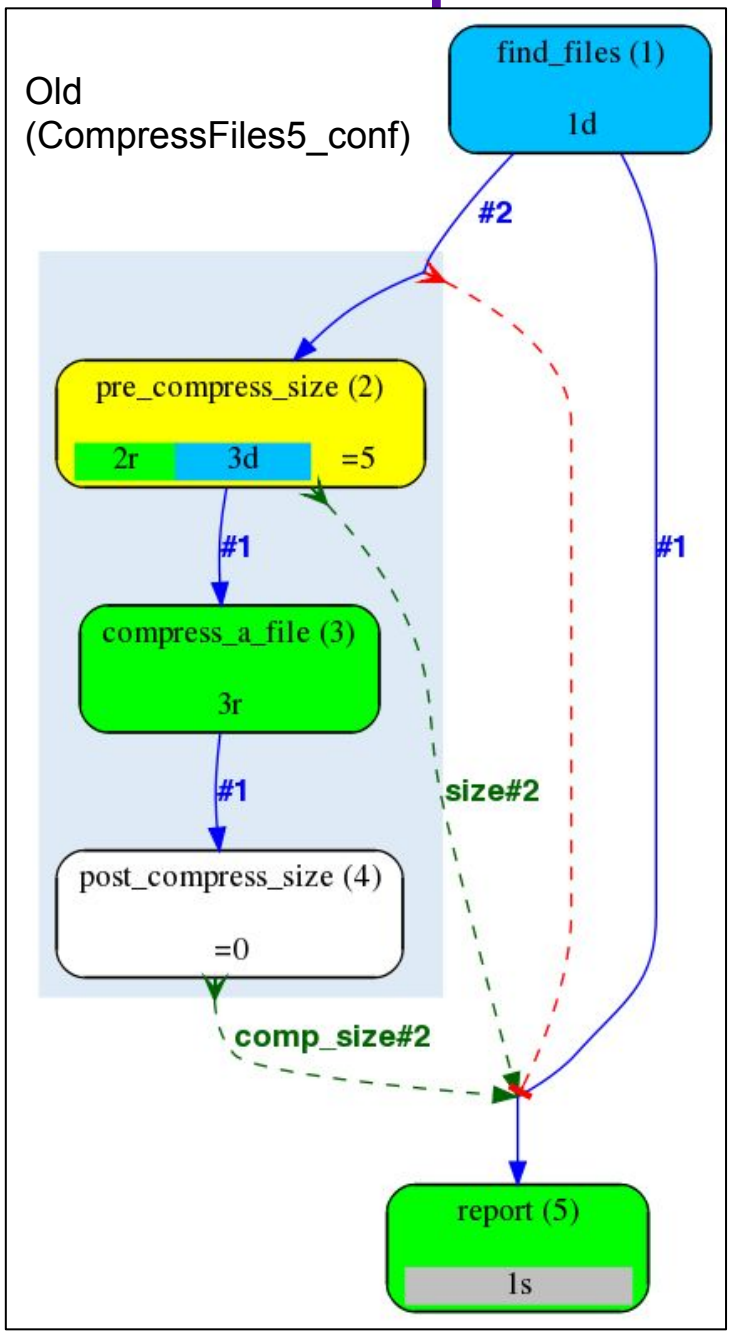
Exercise A: write a simple compress files runnable

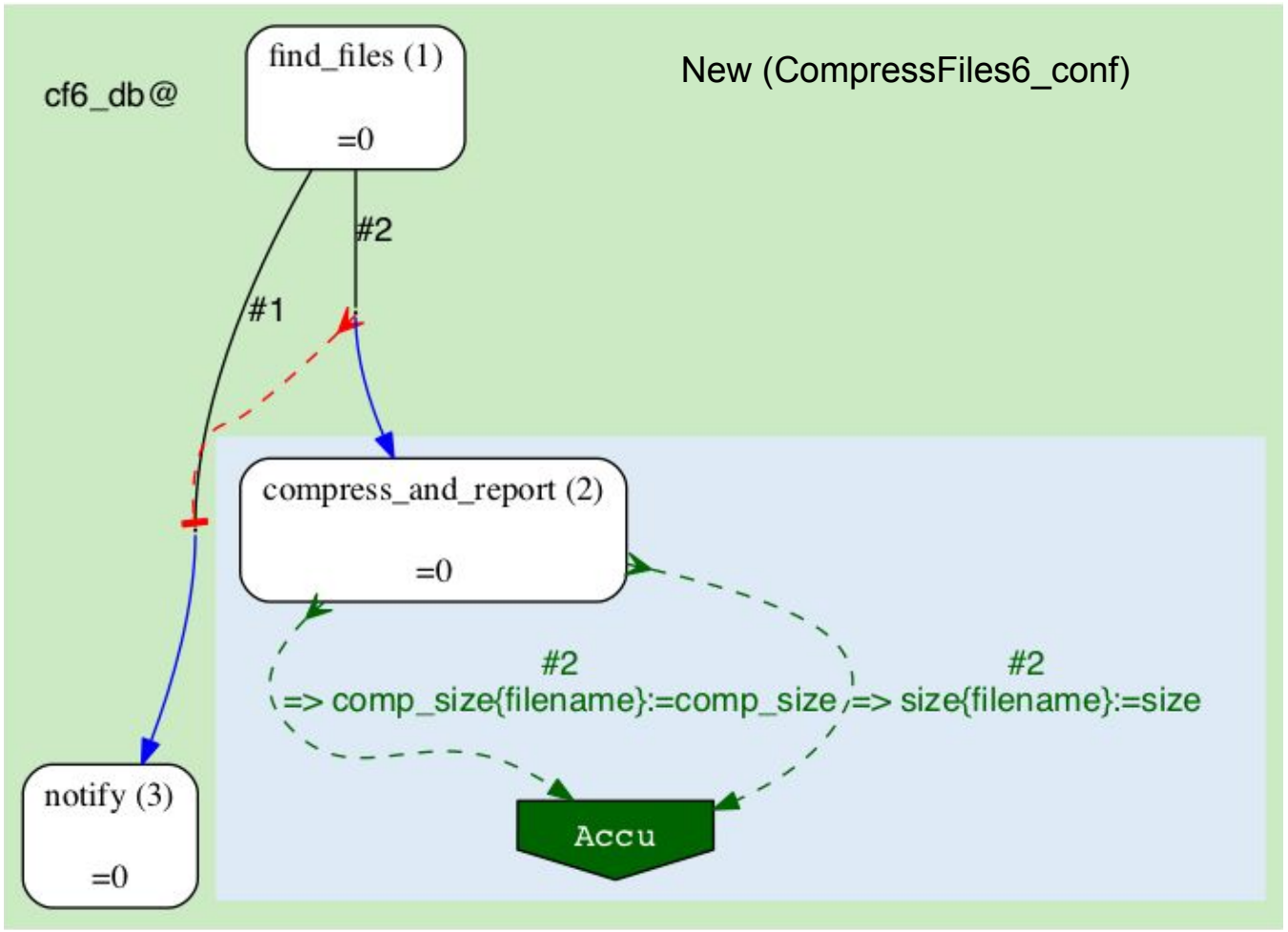
- During the previous section, we built up a pipeline using eHives "standard runnable library"
- In CompressFiles5_conf, we had a section with three steps chained together, operating on a single file (single segment of the fan)
  - Find a file's size before [de]compression
  - [de]compress a file
  - Find the file's size after [de]compression
- Write a runnable to perform all three of these steps in one job

- Hints:
  - See CompressFiles6_conf.pm in solutions4.tgz for a PipeConfig that uses the Runnable you'll be writing
  - Process.pm provides a method called run_system_command that lets you run a command and retrieves the return value, and any messages sent to STDERR:

```
my ($return_value, $stderr, $flattened_command) = run_system_command($command);
```

CompressFiles5_conf vs CompressFiles6_conf





Exercise B: write a simple reporting runnable

- During the previous section, we built up a pipeline using eHive's "standard runnable library"
- In CompressFiles5_conf, the last funnel analysis is supposed to make some sense of the two accumulated hashes sent to it by other jobs

- Write a runnable that gets the data from the two hashes and prints it in a human-readable form. Optionally: find the file with the best compression ratio.

- Hints:
  - Have a look at Hive's Dummy.pm Runnable and SystemCmd.pm Runnable for the general structure of a Runnable
  - Use Hive's parameter API to require and retrieve the two accumulated parameters.
  - Use Hive's $self->warning() function to communicate the results to the user (they will end up in 'log_message' table in the database)

Now you should know everything you need...

- … to finish the exercise.

- Solutions:
  - CompressAndReport.pm (in solutions4.tgz)
  - ReportResults.pm (in solutions4.tgz)

Questions?

http://training.ensembl.org/events/2017/2017-03-23-ehiveRoslin

Ensembl Acknowledgements

The Entire Ensembl Team

Bronwen L. Aken[1], Premanand Achuthan[1], Wasiu Akanni[1], M. Ridwan Amode[1], Friederike Bernsdorff[1], Jyothish Bhai[1], Konstantinos Billis[1], Denise Carvalho-Silva[1], Carla Cummins[1], Peter Clapham[2], Laurent Gil[1], Carlos García Girón[1], Leo Gordon[1], Thibaut Hourlier[1], Sarah E. Hunt[1], Sophie H. Janacek[1], Thomas Juettemann[1], Stephen Keenan[1], Matthew R. Laird[1], Ilias Lavidas[1], Thomas Maurel[1], William McLaren[1], Benjamin Moore[1], Daniel N. Murphy[1], Rishi Nag[1], Victoria Newman[1], Michael Nuhn[1], Chuang Kee Ong[1], Anne Parker[1], Mateus Patricio[1], Harpreet Singh Riat[1], Daniel Sheppard[1], Helen Sparrow[1], Kieron Taylor[1], Anja Thormann[1], Alessandro Vullo[1], Brandon Walts[1], Steven P. Wilder[1], Amonida Zadissa[1], Myrto Kostadima[1], Fergal J. Martin[1], Matthieu Muffato[1], Emily Perry[1], Magali Ruffier[1], Daniel M. Staines[1], Stephen J. Trevanion[1], Fiona Cunningham[1], Andrew Yates[1], Daniel R. Zerbino[1] and Paul Flicek[1,2,*]

[1]European Molecular Biology Laboratory, European Bioinformatics Institute, Wellcome Genome Campus, Hinxton, Cambridge CB10 1SD, UK and [2]Wellcome Trust Sanger Institute, Wellcome Genome Campus, Hinxton, Cambridge, CB10 1SA, UK

Funding

wellcome

EMBL

National Human Genome Research Institute

Open Targets

BBSRC bioscience for the future

TRANSFORMING GENETIC MEDICINE INITIATIVE

MedBioinformatics

Co-funded by the European Union

pipeline_wide_parameters()
and the order of precedence of parameters

The source of parameters is unknown to Jobs

```
sub pipeline_wide_parameters {
    my ($self) = @_;
    return {
        %{$self->SUPER::pipeline_wide_parameters},

        'gzip_flags'    => '',
        'directory'     => '.',
        'only_files'    => '*',
    };
}
```

- Parameters can be:
  - “local” to the Job – accu & input_id (belonging/sent to the Job itself or its “stack” of ancestors)
  - analysis-wide parameters
  - pipeline-wide parameters
  - defaults set in the Runnable’s code

Templates: the other kind of glue

- Runnables have *fixed parameter names* for input and output -
  in comparison with Perl subroutine calls that have a *fixed order of parameters*:
  - + more flexible - you can specify certain parameters and not others
  - + less error-prone - if you add parameters, there is no need to reshuffle them
  - + you may need “glue” to link analyses together

- Two kinds of glue:
  - + input transformation using parameter substitution:

```
'cmd' => 'gzip #filename#'
```

  - + output transformation using templates:

```
2 => { 'compress_a_file' => {
                'input_filename' => '#output_filename#', # rename
                'check_input_once' => 1,                 # specific mode
                'gzip_flags' => '#gzip_flags#',          # explicit propagation
        },
       'another_analysis' => undef,   # no template - use as is
},
```

- Templates work the same way independently of Dataflow’s *destination type*