

What is L-Revolver

L-Revolver is a thread management system for Lasso. It allows you queue up tasks and fire them: asynchronously, as independent queued threads or simply inline.

What use is it to me?

Well, that depends on your programming style and how often you work with (or try to work with) parallel tasks.

L-Revolver removes a lot of the difficulty when working with parallel threads or asynchronous tasks. It provides mechanisms to queue, launch and monitor Lasso tasks.

Tasks can be as simple as sending an email or including a URL, or perhaps more usefully: complex sub-routines running within a complex class or environment.

What is it again?

Think of it as a revolver for your Lasso tasks (what ever they may be), it allows you to load them up and fire them in a number of different ways – all at once*, independently or protected and sequential. This can increase performance and also stability within critical applications. It also allows for unstable tasks to be isolated from the core.

** yes - an actual revolver only shoots one bullet per shot, all analogies eventually fail...*

Features

- **Async Include and Library support** - allows for unmodified includes to run as Async threads
- **Complex parallel task simplification** - easy multi threading without the headaches
- **Thread Isolation** - keep your core running merrily
- **L-Debug support** - www.l-debug.org (ok, that's only a feature for some)



DISCLAIMER: I DON'T PARTICULARLY AGREE WITH GUNS OR THE COLOUR PINK. A REVOLVER SEEMED LIKE A PRETTY GOOD ANALOGY FOR WHAT THIS THING DOES...

Installation

1. Download the package from www.L-Revolver.org/L-Revolver.zip
Or check out with Subversion from [svn://svn.zeroloop.com/L-Revolver/tags/public/stable](http://svn.zeroloop.com/L-Revolver/tags/public/stable)
2. Extract the files to a folder within your wwwroot - we'll use /debug to demo
3. Ensure the .ctyp extension is enabled in Site Admin > File Extensions > Lasso Page Extensions
Or you **can change the extension** to whatever you wish (.inc etc)
4. Type definition. You have a number of options here, for now let's just include it when we need to.

```
include:'/revolver/revolver.ctyp';
```

If you want to get the most out of L-Revolver you will need to define it as a Global type. This can be done by placing the revolver.ctyp file in your site start-up folder. Or more conveniently add the following code to the start of your initialise / controller routine.

```
namespace_using:namespace_global;  
    include:'/revolver/revolver.ctyp';  
/namespace_using;
```

L-Revolver Usage

L-Revolver allows you to stack parallel tasks – we'll start off with the most basic of examples and then begin to build from there.

First we initialise our revolver like so:

```
local('revolver') = revolver;
```

Then we need to load tasks the tasks we want to run like so:

```
#revolver->load( {sleep:500} );
#revolver->load( {sleep:500} );
#revolver->load( {sleep:500} );
```

Then we fire the tasks like so:

```
#revolver->fire;
```

The above is the simplest of examples and serves no real purpose. We'll move on to some heavy real world example later on. For now, here's how we could load up multiple emails to send using the `email_send` tag.

```
// loop for each email to send
#revolver->load(
  -task = \email_send,
  -params = array(
    -to      = 'test@mydomain.com',
    -from    = 'test@mydomain.com',
    -subject = 'my subject',
    -body    = 'Hello World'
  )
);
```

We have a number of options when firing the loaded tasks. Below are the different trigger techniques we can call - these determine the type of thread that each task is run within:

```
// Protected inline tasks
#revolver->fire;

// Launch the entire revolver as a single async thread
#revolver->fire(-async);

// Independently launch threads asynchronously
#revolver->fire(-asyncThreads); // The most common and perhaps should be the default

// Launch independent threads - but launch them in load order (queued)
#revolver->fire(-asyncThreads,-queueThreads);
```

Practical Usage

One common thing that a lot of people find is that they want to include a file(s) within an asynchronous process - Lasso fails in this regard as it does not know the path that the thread was launched within. L-Revolver supports async includes by initialising it with a `-basePath` param like so:

```
local('revolver') = revolver(  
    -basePath = response_localPath  
);
```

One thing to note is that the anon or specified user must have absolute read access to this path in site admin > file paths. ie. read access to: <c://inetpub/wwwroot/mysite> on a windows machine.

You can specify a user for L-Revolver to run as like so:

```
local('revolver') = revolver(  
    -username = 'theUser',  
    -password = 'thisPassword',  
    -basePath = response_localPath  
);
```

Using the above technique will allow you to load includes as tasks like so:

```
#revolver->load( { include('/includes/myReallySlowScript.lasso') } );  
#revolver->load( { include('/includes/myEvenSlowerScript.lasso') } );  
#revolver->load( { include('/includes/whyDoIUseIncludesAgain.lasso') } );  
#revolver->load( '[array]' );
```

And of course fire them all at once: (lets hope they're not hitting a slow datasource)

```
#revolver->fire(-asyncThreads);
```

Or a more conserved: (let each thread finish before launching the next)

```
#revolver->fire(-asyncThreads,-queueThreads);
```

We could also just use `-async` but that would increase the likely hood of each thread interfering with one and another and ideally we don't want that. One of L-Revolver's core strengths is thread Independence.

Back to Reality

I do not recommend that people use includes like in the above examples. I've only included the above examples as I believe that a lot of people may use includes in this fashion.

Where L-Revolver really shines is with custom types and tags (or classes, functions and methods). L-Revolver can execute any function or method asynchronously (and also within any respective class). This is really useful if you have lots of tasks to run with tags that are normally executed inline.

Loading Custom Tags

This works pretty much the same as the email_send example - first we load the tag and along with any params before firing the revolver.

```
// Load each task
#revolver->load(
    -task = \myCustomTag,
    -params = array('hello world')
);

#revolver->load(
    -task = \myCustomTag,
    -params = array('another task')
);

// Fire revolver
#revolver->fire(-async);
```

Member tags (methods) can be run within their respected type (class) by using the -owner param like so:

```
// The below would be run inside of a custom type
#revolver->load(
    -task = #myType->\myMethod,
    -owner = #myType,
    -params = array('hello world')
);

// Or more simply when called within a member tag (method)
#revolver->load(
    -task = self->\myMethod,
    -owner = self,
    -params = array('hello world')
);
```

On The Fly Methods

Using compound expressions we can generate methods to execute within a class like so:

```
// The below could be run within any type
local('myMethod') = {
    sleep:500;
    local('now') = date_mSec;
    log_critical('Type: 'self->type', locals: 'locals');
};

loop:5;
    #revolver->load(
        -task = #myMethod,
        -owner = array,
        -params = array(-someLocal = 'Hello: 'date_mSec)
    );
/loop;

// Fire them independently
#revolver->fire(-asyncThreads);
```

Are We Still Running?

Often we'll want to fire off a stack of tasks periodically check on there progress while our code continues to do whatever we asked of it. We can ask L-Revolver if still has tasks running by using the `->isRunning` method. Below is an example of some code that periodically checks to see if it has any tasks running.

```
local('revolver') = revolver;

// Load some random tasks
#revolver->load({include_url('http://www.google.com')});
#revolver->load({include_url('http://www.msn.com')});
#revolver->load({include_url('http://www.yahoo.com')});

// Fire the tasks
#revolver->fire(-asyncThreads);

// Run some other code
'Do something else';

// This loop will continue while L-Revolver is running
while:#revolver->isRunning;
  // Not the most efficient example
  'Yup, still running';
  sleep(100);
/while;

'Finish Up!';
```

Using `while` is quite intensive - so it's better to use L-Revolver's build in `->wait` tag like so;

```
local('revolver') = revolver;

// Load some random tasks
#revolver->load({include_url('http://www.google.com')});
#revolver->load({include_url('http://www.msn.com')});
#revolver->load({include_url('http://www.yahoo.com')});

// Fire the tasks
#revolver->fire(-asyncThreads);

// Run some other code
'Do something else';

// Until we decide to wait for L-Revolver to finish
#revolver->wait(30000);

// Now we can continue with the L-Revolver results in place;
'Finish Up!';
```

And when using multiple revolvers we can quite easily build up complex interdependent multithreaded tasks.

L-Revolver within Custom Types / Classes

So far most of the examples given have been with inline code. When combined with custom types / classes we can easily pass data between tasks (and threads) by using internal data types. Below is an example class that collates and returns the results of the included URL's - it's a sub-classed array using L-Revolver internally.

```
define_type:'dataBin','array';
  local('revolver') = revolver;

  // queueURL takes a URL and loads it as a Revolver task
  define_tag:'queueURL',-required='url';
    self->revolver->load(
      -task = @{{self->insert(include_url(#url))}},
      -owner = @self,
      -params = @array(-url=#url)
    );
  /define_tag;

  // getData simply fires the revolver and waits for all the results
  define_tag:'getData';
    self->revolver->fire(-asyncThreads);
    self->revolver->wait(30000);
    return(self->parent);
  /define_tag;

/define_type;

local('dataBin') = dataBin;

// Queue some URLs
#dataBin->queueURL('http://www.google.com');
#dataBin->queueURL('http://www.yahoo.com');
#dataBin->queueURL('http://www.msn.com');

// Get the data
#dataBin->getData->size;
```

At this stage I think it important to point out that `include_url` merely serves as a great example call as it normally takes some time to return a result. By no means should you limit yourself to `include_url`. L-Revolver suites any complex sub-routine that can take advantage of multiple threads.

L-Revolver Stacking

In the above example we looked at utilising L-Revolver within a single custom type. L-Revolver instances can also comfortably be stacked within other L-Revolver instances. You simply configure each revolver to suite it's current task and then queue up each revolver as a task within another Revolver. Yes, that sounds complex (and was a mouthful to type) but in reality it's actually simple to put to use effectively. The more you start to think with a parallel mindset the more applications you find that suit this approach.

For a real world example of Revolver stacking (and L-Revolver in general) check out L-Unit (www.l-unit.org) it's core foundation of running tests is built upon L-Revolver. It's also a great example of Async threads being queried in real time.

Querying Threads

Threads can be queried by simply iterating through them like the below example. Currently there's only a limited number of public calls as I think it would be best to keep most of this internal until things mature and public methods can be clearly defined.

```
// Iterate the threads
iterate:#revolver->threads,local('thread');

// Is the thread running
#thread->isRunning;

// Is the thread complete
#thread->isComplete;

// Current status in plain text
#thread->status;

// Current or completed processing time (milliseconds)
#thread->processTime;

// Error infomation
#thread->error_code;
#thread->error_msg;

// Return completed completed info (currently only exists when complete)
#thread->info; // This will also contain any L-Debug stacks

/iterate;
```

Public Methods

These tags will remain constant and are safe to use.

Revolver Type

Member Tag	Params	Description
->onCreate	-username -password -basePath -timeout	User tasks will run as Password for task user Base path for tasks (predominately used by includes) Time to wait for tasks to complete
->load	Loads a task to be run	
	-task (default) -params -owner -timeout	Task to run (generally a tag, can be a string to [process]) Params to be passed to the tag (generally an array) Owner in which the tag will run Specific thread timeout (milliseconds)
->fire	Launches all tasks	
	-async -asyncThreads -queueThreads	Launch entire revolver as async thread Launch each individual task as an async thread Run threads in order (and possibly still async)
->wait	integer	Wait for tasks to complete (milliseconds)
->isComplete		Returns true is all tasks are complete
->isRunning		Returns true any task is still running
->threads		Returns array of current threads (each task is stored within a thread)

Internal Thread Type

Member Tag	Params	Description
->status		Returns current status as text
->error_code		Task error code when complete
->error_msg		Task error message when complete
->processTime		Returns task processing time (milliseconds)
->info		Returns map of information when complete
->debug		Returns existing debug stack when complete
->isComplete		Returns true is task is complete
->isRunning		Returns true if task is still running

De-amble (yeah, it's not a word)

I suspect there's only a few of you that may find this useful (or hopefully at-least interesting). But in today's age where cores are a plentiful resource it makes sense to begin to put them to use when ever possible. Hopefully this tool will help to get Lasso programmers begin to think in a parallel fashion. This is something I've found useful and with a bit of luck, by releasing it out here - someone else may also find it useful.

Feedback - all is welcome, this type of approach is also fairly new to me. Any criticisms, flaws, improvements or even positive comments will be welcomed. Fire them through to ke at the zero loop domain.