NAME

sci - simple continuous integration framework

DESCRIPTION

The **sci** framework is a set of utilities to run automated tasks similarly to alternatives such as jenkins or buildbot.

In contrast to those, **sci** does not know how to build project nor how to retrieve information when to build them. It only works by user project scripts to be executed upon addition of jobs.

It is designed in mind to be as simple as possible to improve flexibility and simple documentation.

OVERVIEW

The **sci** framework is split into three individual programs that are used independently. The communication workflow is:

```
               o ---- scictl
              /
scid (HTTP) o
     |       \
     |        o ---- sciworkerd
   SQLite
```

The **scid** daemon is the unique access to the SQLite database and simply take requests over an HTTP REST API to retrieve and set results into it.

The **scictl** is the administrative utility to update the database using a command line interface. It can also be used to create jobs and their result manually if wanted.

The **sciworkerd** is a daemon executing jobs on a host machine. It access the jobs listing by querying **scid** and output their result.

ENTITIES

The process handles different kind of entities in the database.

PROJECTS

A project is a user description of what to be automated and tested. It has a name, description, project URL and a script to execute. They can be created using the **project-add** command from **scictl**.

JOBS

Jobs are tasks to be performed by any worker for a given project. It has an user arbitrary tag that will be passed to the project script as sole argument. In contrats to many CI system, the sci framework has no information about how to build and access a project and as such the job tag can be anything up to the user (a SCM repo revision, date, simple id, etc).

WORKERS

A worker is a host system that connects to **scid** using HTTP protocol to get acces to jobs to perform, execute them and finally send the result back. They have been designed to use HTTP to allow remote usage.

JOB RESULTS

A job result is the detail about a job ran by a worker for a specific project. If a job exists for one project and there are four workers on the user installation, there will be four job results. It has an exit code (got from the user script), a log console (capture from standard output and error) and a timestamp when it was started.

GETTING STARTED

To setup the **sci** framework you need at least:

1. A daemon scid(8) running and accessible remotely.

2. One or more sciworkerd(8) running on a machine that can access scid(8).

3. Add some projects and register those workers using scictl(8) utility.

Setup scid

The scid(8) daemon does not require much configuration, you can specify the database file to use with its appropriate options. Otherwise, you can simply run the daemon with no arguments. It will initialize the database and generate a default API key that scictl(8) and sciworkerd(8) require to perform requests. Use the **api-get** command to get that key.

The scid(8) program isn't a daemon by itself but a CGI or FastCGI process which needs to be coupled with a dedicated web server. For the example let's use nginx and the kfcgi(8) utility to spawn our FastCGI process.

Configure the nginx server to include at least the following code snippet for a specific virtual host (the scid(8) process explicitly requires its own virtual host).

```
server {
        server_name sci.myhostname.fr;
        listen 80;

        location / {
                fastcgi_param QUERY_STRING      query_string;
                fastcgi_param REQUEST_METHOD    $request_method;
                fastcgi_param CONTENT_TYPE      $content_type;
                fastcgi_param CONTENT_LENGTH    $content_length;
                fastcgi_param SCRIPT_FILENAME   $document_root$fastcgi_script_name;
                fastcgi_param SCRIPT_NAME       $fastcgi_script_name;
                fastcgi_param PATH_INFO         $document_uri;
                fastcgi_param PATH_TRANSLATED   $document_root$fastcgi_path_info;
                fastcgi_param REQUEST_URI       $request_uri;
                fastcgi_param DOCUMENT_URI      $document_uri;
                fastcgi_param DOCUMENT_ROOT     $document_root;
                fastcgi_param SERVER_PROTOCOL   $server_protocol;
                fastcgi_param GATEWAY_INTERFACE CGI/1.1;
                fastcgi_param SERVER_SOFTWARE   nginx/$nginx_version;
                fastcgi_param REMOTE_ADDR       $remote_addr;
                fastcgi_param REMOTE_PORT       $remote_port;
                fastcgi_param SERVER_ADDR       $server_addr;
                fastcgi_param SERVER_PORT       $server_port;
                fastcgi_param SERVER_NAME       $server_name;
                fastcgi_param HTTPS             $https;
                fastcgi_pass unix:/var/www/run/httpd.sock;
        }
}
```

Now, start the process using kfcgi(8). It is recommended though, that the process lives in a clean chroot but in the example we will skip that because scid(8) cannot be built as static binary yet, you could as an alternative setup a chroot where all required libraries are available in order to run scid(8) inside of it.

We will assume that the webserver is running with *www* user and group which is the default also used with kfcgi(8) utility. The webserver must have read/write access to the UNIX socket generated with kfcgi(8).

This command needs to be ran as root but it will drop privileges to appropriate users and groups. See kfcgi(8) for more details.

```
# kfcgi -p / -- scid -f
Or
```

```
# kfcgi -p / -u www -U www -- scid -f -d /path/to/sci.db -t /path/to/theme
Or if you have a functional chroot (scid path is relative to it)
# kfcgi -p /srv/sci -- /usr/bin/scid -f
```

Make sure that scid(8) get read/write access to the default database path if you’re not using the **-d** option. Also don’t forget the **-f** option which indicates the process to run as FastCGI rather than plain CGI.

If the command succeeded, Retrieve the stored key:

```
$ scid api-get
1234567890secretABCDEF
```

Both scictl(8) and sciworkerd(8) understand the same options and environment variables. So let’s set the API key as environment variable for the next chapters.

```
export SCI_API_KEY=1234567890secretABCDEF
```

If you run scid(8) on a machine that is not on the same as the worker, you also need to specify the SCI_API_URL environment variable.

```
export SCI_API_URL=http://127.0.0.1
```

Setup a worker

To register a worker, use the **worker-add** command from scictl(8) utility.

```
scictl worker-add openbsd "OpenBSD 7.2"
```

It is *strongly* advised to run a sciworkerd(8) instance inside a chroot or a virtual machine, remember that it will fetch the script code remotely!

```
$ sciworkerd
Or more secure alternative
# chroot /src/sci /usr/bin/sciworkerd -j2
```

Please make sure to read sciworkerd(8) manual page for more tuning options.

Register a project

Create a project named *hello* with a description, URL, a homepage and a script file to execute. The script code can be of any language but it’s advised that you stick with some interpreted language as many different workers may execute it.

```
$ scictl project-add hello "Hello World" "http://hello.org" "hello.sh"
```

Register jobs

Now, you may want to register a job that will be executed for every worker.

Use the **job-add** command from scictl(8) utility. The *tag* argument is the unique argument that will be passed to the script code. In our case we will assume it’s a revision from a SCM repository.

```
$ scictl job-add hello 67470b67e460
```

And after that, any sciworkerd(8) instance will fetch the job, run it and send the result. It is best used with your SCM to add automatic job when pushing changes.

Note: Jobs that are created before the registration of a worker won’t be executed as it would create a high number of jobs to be performed each time you create a new worker.

SEE ALSO

scictl(8), scid(8), sciworkerd(8)