Contents

1 Abstract

Perl is an ideal language for administration automation:

- powerful language to extract information from external commands
- large number of modules on CPAN
- ability to utilize scripts dynamically depending on environment

But (plain) Perl still has some limitations:

- Reports, Monitoring and scanning systems are very platform dependent at the moment
- joining information from several data sources needs explicit code for each data source
- Report crafting (output) needs to hold up-to-date to the report data structure

Some unifying is required ...

- use DBI as a unified interface to several data sources
  - Unified interface for querying from tables and retrieving results
  - DBD::Sys provides unified interface to system tables
  - DBD::AnyData provides unified interface to structured files (e.g. log files)
  - DBD::CSV and DBD::DBM for more standard formats
  - DBD::ODBC, DBD::postgres, DBD::Oracle, DBD::DB2, DBD::Unify, DBD::mysql etc. to access CMDB backends
- use Template::Toolkit as unified language to control the report generation.
  - lots of plugins (easy to expand the reports using additional features or create new targets)
  - easy to use syntax (even beginners can create reports)
- connect reports and data sources using Template::DBI
- connect configuration and report templates using Report::Generator

2 Architecture

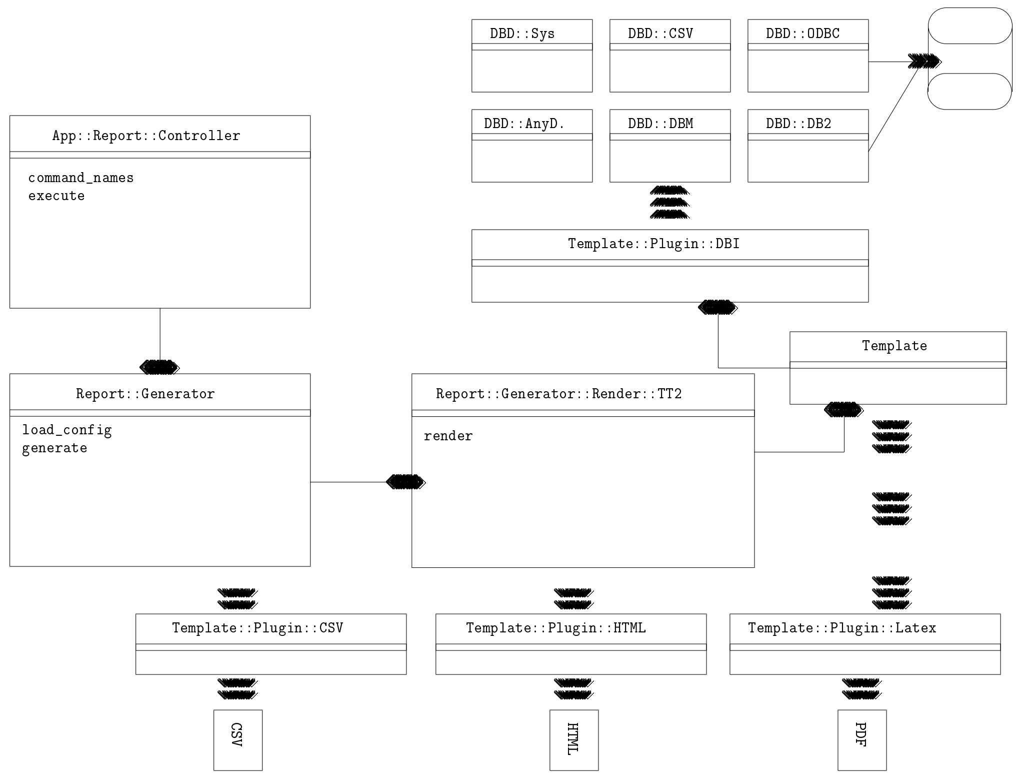

Figure 1: Architecture Overview

3 Modules

Contains short description of some used (but not widely known) modules.

- Report::Generator and it's frontend App::Report::Generator are designed to be as flexible and extensible as possible
- Basic features are provided
- This chapter introduces the default modules which can be used for reporting
- finally some possible extensions are shown

3.1 DBD::Sys

Unified interface to system tables and related data

- Extensible using plugins for new tables
- Plugins are grouped
- Plugins providing the same table extend each other
- Table-Plugins are executed in priorized order

DBD::Sys should be used for real-time analysis of OS states.

```
my $dbh = DBI->connect( "DBI:Sys:", undef, undef, {} );

$dbh->selectall_hashref( "SELECT * FROM procs", "pid" );
$dbh->selectall_arrayref( "SELECT pid,cmndline,username "
                        . "FROM procs,pwent "
                        . "WHERE procs.uid = pwent.uid" );
```

3.2 DBD::AnyData

Unified interface to structured data

- Can read multiple formats using AnyData
- more formats can be added by providing new plugins for AnyData

DBD::AnyData should be used to analyse files provided by applications (e.g. log-files).

```
my $dbh = DBI->connect('dbi:AnyData(RaiseError=>1):');

# define tables
$dbh->func( 'manufacturers', 'CSV', '/users/joe/man.csv', 'ad_catalog');
$dbh->func( 'bikes',         'XML', [$xml_str],            'ad_import');
$dbh->func( 'cars',          'DBI', $mysql_dbh,            'ad_import');

$dbh->selectall_hashref( "SELECT * FROM cars WHERE color='blue'" );
$dbh->selectall_arrayref( "SELECT bikes.* FROM bikes,manufacturers "
                        . "WHERE manufacturers.location IN ('Germany','Austria') "
                        . "AND manufacturers.id = bikes.manufacturersid "
                        . "ORDER BY bikes.serial" );
```

3.3 DBD::CSV and DBD::DBM

Data exports from Excel or OpenOffice.org Calc can easily and quickly be used as database tables using DBD::CSV.

DBD::CSV should be used to enhance the report with some data which are not available on the systems itself (e.g. responsible person contact).

Some binary data files are stored using Berkeley DB (1.x, 4.x) or old dbm format. DBD::DBM can read them all and provide SQL access to process queries on this data.

DBD::DBM could be used to retrieve information from some applications (e.g. pkgsrc packaging system stores some information in Berkeley 1.x database files).

3.4 Report::Generator

Read configuration files, render the report based on the configuration, write the result back and run post-processing actions.

- small, connection utility module
- provides a Template::Toolkit renderer
- reads config files using Config::Any in a lot of formats

```
---
renderer: Report::Generator::Render::TT2
Report::Generator::Render::TT2:
  config:
    ABSOLUTE: 1
    INCLUDE_PATH:
      - share
  options: {}
  output: daily.pdf
  template: report.tt2
  vars:
    ADD_TIMESTAMP: 1
    TARGET: pdf
    FORMAT: latex
```

3.5 App::Report::Generator

Command line interface to Report::Generator using App::Cmd.

- App::Report::Generator is designed to simplify creating reports with short command lines.
- Primary goal is to allow continuous reports over a period of time.

Sample command line and probably sample cron entries (different reports)
With App::Report::Generator it will be possible to

```
genreport gendirect --report /path/to/file
```

This should speed-up report generation, because there is no search for files needed.

3.6 Template::Toolkit

Template::Toolkit is chosen as the environment to render reports for non-developers:

- easy to learn language
- lots of plugins for different target formats
- extensible for advanced users

Template::Toolkit v2 has some disadvantages for power-users:

- automatic dereferencing on some structures
- difficult embedding of callbacks
- nasty surprises in some circumstances

Andy Wardley works on v3 to eliminate them.

3.7 DBI & Other DBDs

Instead of exporting data into files CSV files and work on them using DBD::CSV, one can use DBD::ODBC, DBD::Oracle, DBD::postgres, DBD::mysql, DBD::DB2, DBD::Unify or DBD::SQlite directly.

Template::DBI can handle them all . . .

DBI provides a unified interface to all DBD's, from querying often using prepared statements to fetching results as lists of columns over lists of rows as array or hash reference.

3.8 Other output rendering

While Template::Toolkit is easy to learn and suitable for straight forward easy reports, more complex output generation can be done using e.g. HTML::Mason.

Experienced developers can create their own render classes in Perl. Probably (for creating database import files), SQL::Translator could be interested, too.

4 DevOps

"DevOps is, in many ways, an umbrella concept that refers to anything that smoothes out the interaction between development and operations."

More and more people seem to have huge insight, interest and experience into smart infrastructures. More information on http://www.perl-devops.org/

DBD::Sys is one more way to have a clever infrastructure component.

4.1 DBD::Sys for DevOps

DBD::Sys can help on so much more tasks than reporting:

- smart monitoring
- distributed remote requests with DBD::Gofer
- elegant combining of related information using joins

How to extend DBD::Sys?

1. Decide whether to use an existing plugin namespace:
   - DBD::Sys::Plugin::Meta
   - DBD::Sys::Plugin::Any
   - DBD::Sys::Plugin::Unix
   - DBD::Sys::Plugin::Win32
2. create table module
3. make || ./Build
4. ready to use

How to speed up DBD::Sys? Use one of following tunables in the dbh:

- sys_only_loaded_plugins
- sys_only_loaded_tables

```
$dbh = DBI->connect( "DBI:Sys:", undef, undef, {
    sys_only_loaded_plugins => 1,
    sys_only_loaded_tables => 1, } );
```

5 Examples

We're here to see what can be done with Perl to avoid monolithic scripts like cfg2html. So let's first see how we can use DBD::Sys to monitor some events on our systems and second how to generate a daily report.

5.1 Monitoring

Open example scripts/mon.pl

We have a number of daemon processes we want to monitor for cpu, file, memory usage etc. When each daemon starts it adds it PID and name to a file and when it exits it removes the pid/name from the file.

Another daemon runs monitoring the file with PIDs,names. It spots changes in the file by doing an MD5SUM on the file so if a new daemon starts or ends it spots it. For each pid in the file it uses DBD::Sys to gather stats on the daemon and writes it into tables in the database. Lets say the tables are "procs" (holding PID, name, cpu, memory etc) and "files" which records a row for every file open (via lsof) and the process ID. This daemon loops every N seconds checking for new daemons starting/stopping in the pid file and writing records into "procs" and "files" tables.

So at this stage we have something recording our daemon data into "procs" and "files" tables.

We decide on what our warning limits are - e.g., CPU more than n% or memory more than N megabytes, or more than N open files.

We have an "alerts" table where we write the process id and type of alert if something exceeds a limit.

We create an insert trigger on the "procs" table which does this:

- every time a row is written to the "procs" table the trigger fires
- it examines the cpu, memory etc and if a limit is reached it writes the PID and description of the exceeded limit to the "alerts" table

We have an insert trigger on the "files" table which does this:

- every time a row is written to the "files" table is does a "select count(*) from the files table where pid = N" to find how many open files that process has. If it exceeds the limit we write a row to the "alerts" table with the PID and description of the exceeded limit.

So far, we are recording data on PIDs and when a limit is reached we are writing rows into an "alert" table - good.

Another trigger is added to the "alerts" table. It is an insert trigger so it fires every time something inserts into the "alerts" table. This trigger issues a dbms_alert something like this:

```
create trigger alert_i1 after insert on alert
for each row
begin
 IF INSERTING THEN
    dbms_alert.signal('LIMIT REACHED',
                      TO_CHAR(:new.pid) || ' ' || :new.description);
 END IF;
end;
```

This causes a dbms_alert to be sent whenever a row is inserted into the "alert" table.

Another daemon on the outside of the database connects to the database and issues a dbms_alert.waitany which blocks until an alert is thrown by oracle (i.e., it blocks until a row is inserted into the "alert" table). When it unblocks it has the alert string 'LIMIT REACHED' and the description so it knows what sort of alert was sent. It can split the pid and description out of the alert string and know what happened. It can then send an SMS or email or even kill the offending daemon - whatever you want really.

I could provide more code but I doubt I'd have time. I think it might be a good(ish) example for you. If you have questions or want me to expand on anything you've got about tomorrow and maybe Saturday depending on how much preparation I need for my holiday.

5.2 System Report

```
[% PROCESS html.tt2 -%]
[% USE DBI -%]
[% USE Dumper -%]
[% USE dbh = DBI( 'DBI:Sys:' ) -%]
[% st = dbh.query( 'SELECT * FROM pwent' ) -%]
[% r = {
    cols = st.get_colnames()
    data = st.get_all_list()
    name = 'pwent'
    caption = 'Users on this System'
    } -%]
[% htmltable(r) -%]
[% st = dbh.query( 'SELECT * FROM grent' ) -%]
[% r = {
    cols = st.get_colnames()
    data = st.get_all_list()
    name = 'grent'
    caption = 'Groups on this System'
    } -%]
[% htmltable(r) -%]
[% st = dbh.query( 'SELECT * FROM filesystem JOIN filesysdf USING (mountpoint)' ) -%]
[% r = {
    cols = st.get_colnames()
    data = st.get_all_list()
    name = 'filesys'
    caption = 'Filesystems and free space in them on this System'
    } -%]
[% htmltable(r) -%]
[% st = dbh.query( 'SELECT * FROM netint' ) -%]
[% r = {
    cols = st.get_colnames()
    data = st.get_all_list()
    name = 'netint'
    caption = 'Network interfaces on this System'
    } -%]
[% htmltable(r) -%]
```

6 Open Points

- requires local installation
- nearly no application sensors (DB Users, virtual servers in apache, ...)
- SNMP support for generic data access (local and remote)