ROME: R-Omics Made Easy. Developers' Guide

Cass Johnston

December 4, 2006

Abstract

R and the Bioconductor suite of bioinformatics tools provide an extremely useful framework in which to analyse transcriptomics data. Unfortunately the command line interface to R has prevented their widespread use among biologists. With the rapid development of high throughput molecular biology techniques for -omics studies in recent years, biologists have an increasingly urgent need for statistical methods which are designed to cope with these new large and noisy datasets. In order to make the bioconductor tools and in-house R code for transcriptomics analysis available to the biologists with whom we work, we need a relatively quick and easy way of generating graphical user interfaces (GUIs) for them. Hence ROME (R-omics Made Easy).

Contents

Chapter 1

Overview

1.1 Design Decisions

Web-Application

A web-based client-server design was chosen over a standalone piece of software for two main reasons:

Firstly, running the software on a server means that the end users do not have to be concerned with keeping their software up to date. This is particularly important because both R and Bioconductor are under constant development and so must be updated frequently. As R and Bioconductor are modular it is also simpler to ensure dependencies are met on a central server than it is on a number of standalone installations. The alternative would be to take a snapshot of R and Bioconductor and distribute that as part of a monolithic application. Unfortunately, as many of the newer modules, to which the users require an interface, are dependent on the most recent stable versions of R and Bioconductor, a snapshot would rather defeat the point.

Secondly, many of the analyses are computationally intensive. A standalone application would have to be run on a machine capable of processing, for example, large microarray experiments. This is not the type of processing power that biologists typically have to hand on their own desktop. Given this fact, most labs would presumably have to buy a dedicated box on which to run the analysis, in which case it makes more sense to have users connecting to this server as clients and running their analysis than physically sitting in front of the machine.

A client-server model will also make it easier to extend the capacity of the analysis software by having the server delegate the processing to a computer farm.

Perl

Perl was initially selected for compatability with the European Bioinformatics Institute(EBI)'s Expression Profiler (EP) tool. After a few months of development it became clear that the low-level analyses we were interested in simply didn't fit well into the EP data model (which focuses more on clustering and dimension-reduction algorithms for multivariate analysis), so it made more sense to develop ROME seperately. That said, much of the code developed for EP was still of use, so it made sense to continue with Perl. Perl is a popular choice in bioinformatics applications due to its suitability as a glue language, its strong support for text parsing and the number of useful modules on CPAN (Comprehensive Perl Archive Network). This will make maintenance of ROME easier in the long term.

Catalyst

To begin with, ROME was developed from scratch, doing all its own session management, user management, access privileges and other basic framework functions. This was time-consuming to develop and debug. It was also my first development project and as such a steep learning curve. Much of the code written at the start of the project was less than ideally modular and easy to maintain. Eventually the decision was made to migrate to the new Catalyst Model-View-Controller (MVC) web-development framework (Riedel, 2006). Although this process took about a month, subsequent development was much faster as it left only the ROME-specific problems - managing and describing experiments, generating R scripts and so on. Catalyst provides the user and session management, and automates much of the basic CGI processing (extracting form values, URL mapping etc). Catalyst is also rapidly increasing in popularity and has many similarities to the already widely used Ruby-on-Rails framework, so finding people able to maintain ROME in the future will be much easier than if the entire application was custom built.

Open Source

There are already a number of commercial software packages available for microarray analysis, including Agilent's GeneSpring and Stratagene's ArrayAssist. Clearly there is little point in developing a package in direct competition with these already successful tools. The aim of ROME is to provide an open-source solution. There have been many long debates on the relative merits of the open and closed source development models (Raymond, 2006; EU, 2005). There are obvious financial benefits to using free software, but this is not a primary concern as the sums of money involved in software acquisition are generally dwarfed by the costs of the necessary lab equipment for transcriptomics studies. Of greater potential benefit is the speed at which new algorithms could be made available to users with an open source development model. There has been a significant lag between the adoption of a method as *de facto* standard in Bioconductor and the integration of that method into commercial software. The Bioconductor community has already demonstrated that it is capable of rapid production of useful and successful analysis algorithms using an open source model as seen in the previous chapter. It is hoped that they will be as proactive in developing interfaces to these tools once supplied with a framework to do so.

1.2 Catalyst

ROME was built with Catalyst (see section 1.2), a perl web developement framework. Data is stored in a MySQL database and statistical analysis is performed in R using the Bioconductor tools and some custom code. It has been designed such that adding new R scripts is a relatively simple process, allowing biologists to have rapid access to new bioconductor developments.

A basic understanding of the framework on which ROME is based is required to understand the system. Catalyst (http://catalyst.perl.org) is a Model-View-Controller (MVC) web-framework written in Perl. MVC is a widely used standard pattern for designing web-applications.

Figure 1.1: Model-View-Controller Pattern

The model,view and controller tiers can be considered as a variation on the more traditional input-processing-output paragdigm. The model is the processing layer, in the sense that it encapsulates the available data and the processes one can perform upon that data. The controller layer is analogous to the input as it deals with user requests, decides where to send them and asks the model to make appropriate changes or return appropriate information. The view deals with presentation of results to the user.

M

The model tier deals with information processing, retrieving and altering information stored on the system. Typically, this will involve interacting with a database. There are a number of well-established Perl modules that provide a mapping between relational databases and perl objects, based on the standard DBI Database Interface module. These include Class::DBI and its extension Class::DBI::Sweet and the newer DBIx::Class. Catalyst has Model modules which are effectively just wrappers around these database mapping modules that plug them into the catalyst framework

V

The view tier deals with presenting results to the user. For web development, this generally involves a templating system of some sort to insert the dynamic content into the static HTML outline of each page. Again, there are many modules on CPAN which provide templating functionality, including Mason, HTML::Template and the Template Toolkit. There are Catalyst wrappers to allow you to use all of these and more within the Catalyst framework.

C

The controller deals with handling the application flow. In other words, it takes requests from the user (from the view), determines what to do with them, tells the model to make the appropriate alterations to the data, and passes on any results back to the user. The Controller layer is the heart of Catalyst and where it does most of the work. Model and View functions are mostly supplied by plugin modules, but in the Controller layer Catalyst does all the heavy lifting.

The aim of the MVC pattern is seperation of concerns such that it is possible to change one layer of the application without affecting the others, for example changing the way in which the data is presented to the user via the view, or reusing the model in different applications. Generally an event causes a controller to change the model, the view or both. Changes to the model are automatically reflected in any dependent views. A more in depth discussion of the MVC pattern can be found in Gamma et al. (1994).

Catalyst processing

Catalyst accepts http requests and forwards them on to one of the controllers as determined by the URL. The controller method will then take the request, retrieve or modify data in the model as requested and then return some result to be passed back to the user. Finally the controller will forward the result to the view. Catalyst determines which Controller is responsible for processing the request by URL mapping. For example the URL:

```
MyApp/Test/do_something/93
```

might call the `do_something` method of the Test controller with the argument 93. See section 1.2 for more details of URL mapping.

Catalyst automatically creates a context object which is available everywhere in the application. This context object contains Catalyst::Request, Catalyst::Response, Catalyst::Config and Catalyst::Log objects (details of which can be found in their perldoc on http://search.cpan.org). This provides the controller with access to information about and content from the http request, details of the application configuration and a log for messages. Generally the response is handled by the view, but the controller may need access for setting cookies, redirects and so forth. The context object also contains a hash known as the stash, which provides place to store data to be shared between application components, primarily between the controller and the view.

1.2.1 Catalyst Controller Actions

A Catalyst controller Perl module has *actions* rather than *methods*. An action is effectively just a method which has a rule defining the URLs to which it should be mapped. Catalyst has various types of actions, which differ in their URL matching rules:

Literal `sub foo : Path('/bar')` which match relative to their current namespace

Regex `sub foo : Regex(^item(\d+)/order(\d+)$)` which match any URL that matches the pattern. The match is not relative to the current namespace.

RegexLocal `sub foo : RegexLocal(^widget(\d+)$)` which match any URL that matches the pattern. The match is relative to the current namespace.

Global `sub foo : Global{}` in which case the function name is mapped directly to the application base, for example http://localhost/foo

Local `sub foo : Local{}` in which case the function name is mapped relative to the current namespace.

Full details of Catalyst's URL dispatching rules can be found from the Catalyst::Manual perldoc (web).

It is also possible to define `Private` methods which are not mapped to a URL and may only be called internally using the Catalyst context object's `forward` or `detach` methods. Catalyst has a number of pre-defined private actions, called in pre-defined circumstances, that can be used or overridden in controllers:

default : Is the method called when nothing else matches. Generally this would be used to set a 404 (page not found) error back to the client. It can be defined in the base application module and will apply to other controllers unless specifically overriden.

index : Is the method called when the URL matches the base controller namespace.

begin : Called at the beginning of a request before any matching actions

end : Called at the end of a request

auto : Called at the beginning of a request and must return true for processing to continue.

A flowchart of the Catalyst request cycle are shown in figure 1.2.

Figure 1.2: Catalyst Processing Flow
dev.catalyst.perl.org

1.3 Sessions

In order to store information between http requests, the Catalyst Session plugin is required. The session plugin adds a session to the the Catalyst context object. Data stored in the session hash is stored on the server and linked to a session key. The session key is also stored client-side, typically in a cookie. The client then sends the session key data along with each request and any data associated with that session key is restored to the session hash. The Session plugin requires two backend plugins to deal with the mechanics of storing data on the server and maintaining the session state on the client. There are a number of options in Catalyst, but ROME uses Session::State::Cookie and Session::Store::FastMmap.

1.4 Error Handling

1.4.1 Internal Server Errors

In `lib/ROME/Controller/Root.pm` If an exception occurs in a controller then the client either receives a suitably apologetic error page or, if the debug flag is on, a debug page with various bits of useful information. You can set an error message using `$c->error($msg)` but this will only be displayed to the user if debug mode is on.

You can use `$c->log` to send information to the log if you wish. See the Catalyst::Log perldoc for details.

1.4.2 User Errors

Ever template automatically gets an error_msg field inserted before any other content (from `root/src/site/layout`) into which you can put any messages you like by simply setting `$c->stash->{error_msg}` in your action. If you only want to display the error and no other content, you can use the `user_error.tt2` template, which is just an empty page with the title error.

1.4.3 Form Validation Errors

Assuming you're using the Data::FormValidator plugin then you can include the dfv_error.tt2 template prior to your form (see user/login.tt2 for an example). This template will format the error messages from the Data::FormValidator::Results object in `$c->form` for you, so if your validation fails, you just return the form template and any errors will appear at the top (see the `login` action in `lib/ROME/Controller/User` for an example)

Chapter 2

Installation

2.1 server

2.1.1 rome_server.pl

2.1.2 Apache

Chapter 3

Data Model

Chapter 4

Core Components

NOTE: All URIs given are relative to your ROME base href, which if you're using the inbuilt testing server will be something like http://localhost:3000/ (more details of how to set up your server can be found in section 2.1). All file paths are given relative to your ROME directory, which will be wherever you put it (see section 2 for installation instructions).

4.1 Navigation

The ROME menu is structured as a nested list of links, styled by css. For browsers which support css hovers, it will work fine without any javascript. For browsers without hover support, it will use javascript for rollovers. For older browsers, if should degrade to a nested list of links.

Changing your menu

The ROME navigation menu is defined in the configuration file ROME/nav.yml. This makes changing your menu structure simple.

The easiest way to see how to change your menu is to simply take a look at the nav.yml file. It should be fairly self-explanatory. A top level menu section would look something like:

```
- Start Here:           # top level menu title, can be anything you like
    title: Start here   # html 'title' field, appears on mouseover
    dropdown:           #start submenu
      - login:                    #a page link. Must be a ROME component name
          display_name: Login     #optional display name
          title: Login to ROME
          href: /user/login       #url, relative to your ROME base
  any_datatype: 1         #disable type-checking for this URL
      - logout:
          title: End your ROME session
          href: /user/logout
  any_datatype: 1
```

```
      - register:
          title: Create a ROME account
          href: /user/register
  any_datatype: 1
```

As it is written in YAML, indentation is important. Use spaces not tabs and your spacing must be consistent. More information about YAML can be found at http://www.yaml.org if required. If in doubt, copy and paste an existing menu section and modify it to suit your needs.

The menu and submenu entries can be named anything you like (though relatively succinct is better). You can use the title fields to add a mouseover tooltip. The links to pages must be named with the name of the ROME component to which they refer, otherwise they'll remain greyed out when a datafile of a type they can use is selected.

By default, ROME compares the datatype of the current datafile with the process_accepts datatypes of the components in the menu and greys them out if they're not usable with this datafile (for more details, see section 4.2.1). For pages which don't process datafiles (for example, most of the core ROME functionality - login, user management, datafile management and so on) you'll need to disable this behaviour by setting the any_datatype flag to a true value.

If you want to prevent bits of the menu from being displayed to users without certain roles, just add a list of the roles who are allowed to view it, something like:

```
- Admin Tools:
    title: Administration tools
    roles:
      - admin
      - dev
    dropdown:
      - manageroles:
          display_name: Manage Roles
          title: Manage roles
          href: /role
```

Note that hiding parts of the menu from certain users is merely a aesthetic choice, it provides no real access control as the user can still type the URL in if they know it. You need to check the user roles in the controller for actions which require authorization (see section 4.2.1

If you make changes to the nav.yml file, run script/rome_makemenu.pl to regenerate the menu template file.

The nav template file

The resulting template file is processed each time the menu is returned. Though this adds a bit of overhead to each call, this is outweighed by the benefit of having a context-aware menu. Any menu items which are irrelevant for the currently selected datafile will be greyed out. A bit of javascript prevents them from even being clickable.

4.2 Users

R-OME requires that you have a user account. This section will look at how users are registered and logged in to the system, how their access to particular ROME components is controlled and how their user accounts are managed.

The user component consists of the controller in lib/ROME/Controller/User.pm and the various template files in root/src/user/.

Authentication

ROME uses the Catlyst plugin Authentication, which adds login and logout methods to the catalyst context object.

The Authentication plugin needs to be able to store and access information about the users. There are various options for doing this in Catalyst, but ROME uses the Authentication::Store::DBIC plugin. In the authentication section of the rome.yml config file we tell the Authentication plugins to use the ROMEDB::Person class and specifically the username and password fields for authentication. The ROMEDB::Person class was defined as part of the model, described in section 3.

The Authentication plugin also needs the Authentication::Credential::Password plugin to provide the necessary logic for checking the username and password with those in the database. For security, the passwords are encrypted in the database using the SHA1 algorithm (which requires the Digest::SHA1 module from CPAN). New passwords are encrypted when the user registers (see section 4.2. The password_type and password_hash_type settings under authentication in the rome.yml config file tell the Authentication plugins to use the appropriate encryption algorithm.

In the controller module lib/ROME/Controller/User.pm, the index action simply hands over control to the login action, which means that the login page can be accessed through either /user or /user/login URLs.

The login action checks for a submit parameter from the form data in the catalyst request object. If it doesn't find any, it sets the template to user/login.tt2 and returns (handing over control to the view). The client is therefore sent the login form.

If the form has been submitted, the values of the form parameters are validated using the Data::FormValidator plugin. user/login.tt2 includes dfv_error.tt2 which checks whether there are any error messages from the form validation and if there are, displays them. If validation fails, the client will receive the login page with any validation errors listed at the top.

Assuming the parameter validation is successful, the username and password parameters are handed over to the login method in the catalyst context object. The login method checks the username and password against those stored in the database and if they match stores the user object in the catalyst context. You can test if a user is logged in with `$c->user_exists`.

Although the passwords are encrypted in the database, they are sent as plaintext between the client and server. To improve security, ROME uses the catalyst RequireSSL plugin. Both the login and registration methods have a call to the require_ssl method which, in a production environment, would force redirect to a secure server. This behaviour is disabled in the Catalyst test server script/rome_server.pl. Configuration settings for the RequireSSL plugin are in rome.yml under require_ssl.

The Authentication plugin also adds a logout method to the catalyst context. This is called by the user/logout action.

Registration

The registration action checks for a submit parameter from the form data in the catalyst request object. If it doesn't find any, it sets the template to user/register.tt2 and returns (handing over control to the view). The client is therefore sent the login form.

If the form is being submitted, the registration action forwards to a private action which validates the parameters using the Data::FormValidator plugin. If validation fails, the form is returned. As the `user/register.tt2` template includes the `dfv_error.tt2` template, any errors arising from the validation will appear at the top of the form.

Once the form parameters have been validated, the action forwards to another private action `_process_registration`, which in turn forwards to `_insert_user`. The `_insert_user` method creates a person and an associated person_pending entry in the database and stores the associated DBIC object in `$self->user`

Once the new user is in the database, the `_process_registration` action checks the config file to see if we need to get confirmation from the user and/or the administrator in order to complete the registration process. If confirmation is not required, the admin_approved and user_approved fields in the new user (actually in the new person_pending, but these are proxied to the person class) are set to 1. If confirmation is required, it forwards to the `_user_email_confirmation` or `_admin_email_confirmation` actions to email requests for confirmation.

Finally the `_process_registration` action forwards to the `_complete_registration` action. This action checks to see if the admin_approved and user_approved fields are both set to true. If they are, it deletes the person_pending entry from the database, leaving a valid new person entry and setting a template which informs the user they can now login. If the approved fields are not true, it sets an appropriate template explaining what to do next. `_complete_registration`

If confirmation emails were sent, they will include a URL to the admin_confirm or user_confirm actions with the username and email_id as parameters. The email_id is just a randomly generated string. The email id in the database must match that in the URL, which acts as a check to ensure that the new user really owns that email address with which they have registered. The admin_confirm and user_confirm actions simply set admin_approved and user_approved to 1 for this user the forward to the `_complete_registration` action.

4.2.1 User Directories

The `_complete_registration` action also creates two directories for the new user: A data directory for their datafiles, which only need to be available to the server and a static directory for any downloadable data they generate - this includes images and export files.

The location of the data directory is specified by the userdata entry in rome.yml. It must be specified as a full path and it must be readable and writeable by the user under which the ROME server is running. Users are given a directory under this location named with their username. A subdirectory called uploads is also created, into which they can upload raw data. This will be discussed in more detail in section **??**

The static directories are created under root/rome/static/user. Each user is given a directory named by their own username which is only accessible by them. It is also possible to create other static directories with different permissions, discussed further in section 4.2.1

Authorization

There are two main types of access control in ROME. One is the URL bases authorization methods provided by the Catalyst authorization plugins, `Authorization::Groups` and `Authorization::ACL`, which controls access to actions. The other is the user and workgroup controls implemented specifically for ROME to control access to datafiles.

Authorization::Roles

This plugin allows you to limit access to a given action to those users with a given role using something like `$c->check_user_roles('admin')`. The plugin uses the person and role tables in the database, via DBIx::Class, as defined in rome.yml under authorization.

Authorization::ACL

There are some cases when you need a more complicated access definition, and for these we use the ACL plugin. The plugin provides Access Control List style path protection. The ACL plugin only operates on the Catalyst "private namespace", which basically means you can only use if for 'Local' actions, it won't work on Path, Regex or Global actions. ACL rules are defined in the `lib/ROME.pm` file. Details of how to define the rules can be found in the Catalyst::Plugin::Authorization::ACL perldoc.

Access controls for data

Datafiles can be publicly accessible, shared with members of a workgroup or private to their owner (the user who created them). Access can be set at the experiment or individual datafile level (with datafile access rules overriding the controls on the experiment to which they belong). This is dealt with in the model and is discussed in more detail in sections **??** and **??**.

User Account

Users can manage their accounts from the URL `user/account`.

User Administration

Chapter 5

Styling

The css file for ROME lives in root/src/ttsite.css

Chapter 6

Scripting

ROME should be usable with javascript disabled, but only for a fairly charitable definition of usable. Much of the user-interface widgetry will be disabled without javascript and there is little benefit to using ROME over an R command line interface without it.

Chapter 7

Testing

7.1 Unit Testing

Uses Catalyst::Test.

7.2 Application Testing

Uses Test::WWW::Mechanize::Catalyst. See all of the `t/live_app*` test scripts.

Bibliography

Comprehensive perl archive network (cpan). URL `http://search.cpan.org`.

EU. Eu report on free, libre and open source software(floss). 2005. URL `http://www.infonomics.nl/FLOSS/report/`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

Eric Raymond. The cathedral and the bazaar. 2006. URL `http://www.catb.org/ esr/writings/cathedral-bazaar/cathedral-bazaar/`.

Sebastian Riedel. Catalyst. 2006. URL `http://catalyst.perl.org`.