Documentation for the RAxML EPA Webserver

Denis Krompass
dekromp@googlemail.com

January 4, 2012

Contents

1 Application concept & structure

The Web-Interface for the RAxML Evolutionary Placement Algorithm is written in Ruby using the Ruby on Rails Framework for web-applications. I recommend reading the first chapters of a Ruby on Rails book (e.g. Agile web development with Rails) or at least an online tutorial to understand the basic ideas and concepts of the framework. There is some "magic" happening in the background of a Rails application that cannot be seen within the scripts and programs that are described here. So somebody who has no idea about these functionalities provided by the framework might get confused about how this collection of scripts can actually work when looking solely at the code.
In this documentation I will try to explain how the scripts and programs are connected and work together. Besides that I will try to give an overview of the jobs each script fulfils to ease debugging or developing new functions within the application in the future. However, I will not provide any explanations to any programming details here. The details can be discovered directly by looking at the code. Some words about my code and programming style respectively. I always try to use meaningful names for variables, methods and classes that explain their content or their functionality. Therefore I often resign using comments where in my opinion the naming is enough for the understanding. An overview of the concept of the web-application is given in figure 1. In the following sections this concept will be explained. Starting by the views and digging from there deeper into the application.

1.1 The folder hierarchy

In this section I will talk a little bit about the structure of the application. When looking in the root directory of the Rails-Application (`RAxMLWS/`), there are a couple important folders:

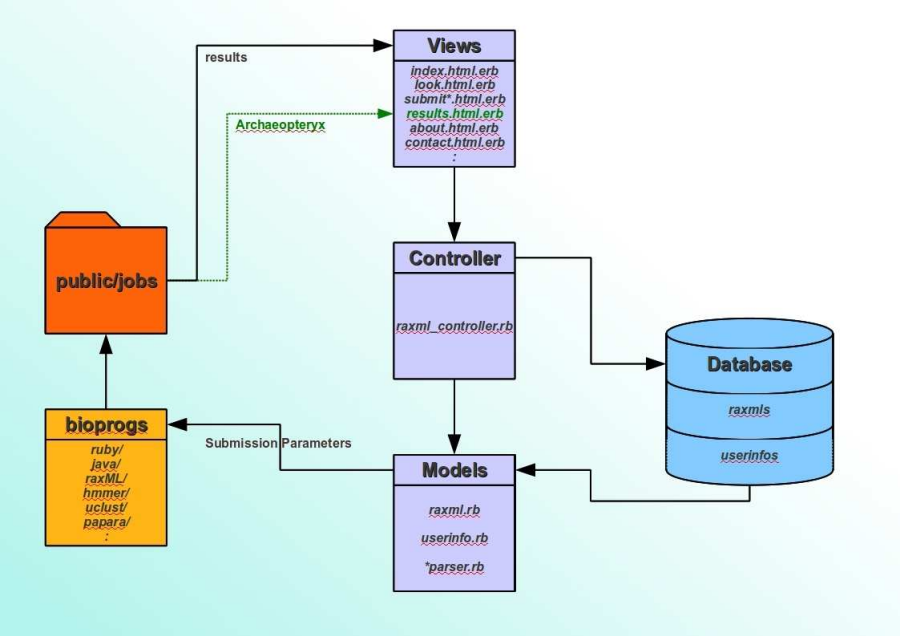

Figure 1: The basic concept and structure of the RAxML-EPA Webinterface

`app`	This folder contains all the models, views and the controller scripts used. The details of these files will be discussed in the next sections.
`bioprogs`	This is an "artificial" folder not provided by the default rails framework. It contains different helper scripts and programs (e.g. RAxML) used by the web-application.
`config`	The rails configurations. Some relevant details are also discussed in a later section.
`db`	Here are the database details provided. Unfortunately rake db::migrate does not work when altering the models and tables that are already present in the database. This has to be done manually by hand in SQL.
`log`	All the rails logs are stored here.
`public`	The public folder contains all kinds of stuff including things the Apache can access like the Archaeopteryx applet and images, and also JavaScript-Scripts and Stylesheets used by the views. This folder also contains Job folders from the users including the job specific results and input files.
`test`	The application has a test-routine that tests some use-cases. Testing is covered in the "Testing" section.
`testfiles`	This folder contains test-files that simulate use-cases for the test-routine.

1.2 Views

Views are the user-interaction-interfaces of the web-application. In this case html.erb pages ("erb" is a synonym for "embedded ruby"). All the views have a common layout which is defined by `app/views/layouts/raxml.html.erb`. The navigation bar (figure 2 green) is defined by `app/views/shared/_navigation.html.erb`. The content (figure 2 blue area) is variable and defined by the views stored in `app/views/raxml/`. In most of the cases the views only define the structure of the web-page. The appearance and some extra functionality is defined by an external CSS-Stylesheet (`raxml.css`) and an external JavaScript script (`application.js`). These files can be found in `public/stylesheets/` or `public/javascripts/` respectively.

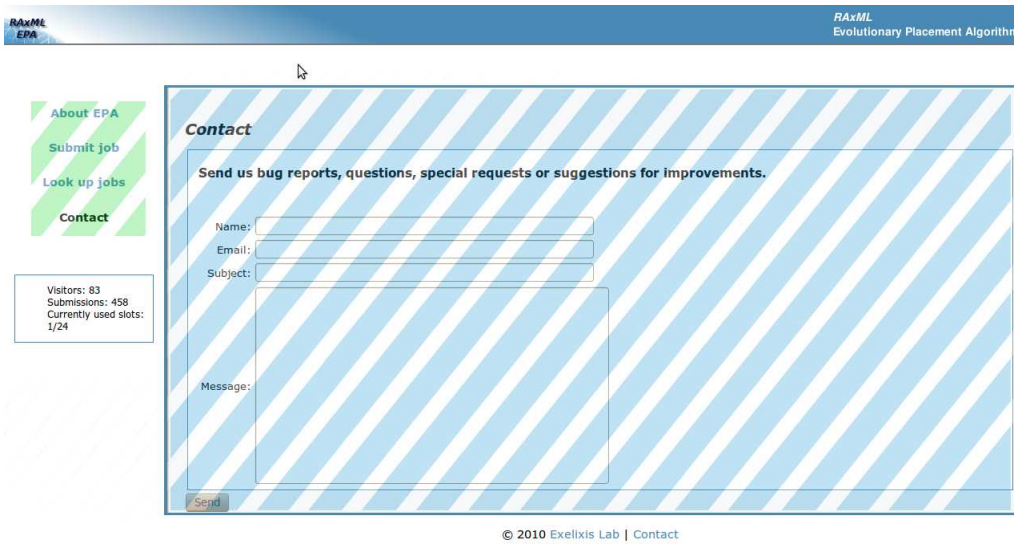

Figure 2: The Webpage Layout

1.3 Controllers

As denoted by their name, controllers control the actions performed within the view. Basically all views in `app/views/raxml/` have a method with the same name defined in the controller, which is executed before the associated view is interpreted. This is useful when you need to display data from an external source within the view (e.g. from a MySQL database). Rails allows you to define global variables within the controller methods that can be accessed within the view. For instance if you want to list all jobIDs for a certain user in your `yourExamplePage.html.erb` , you can do so by loading the desired data within a controller method called `yourExamplePage` from the database and save it in a global variable (marked by an @, e.g. @jobIDs). It is then possible to read this variable within the view using embedded ruby (denoted by the "`<%= @jobIDs %>` tags. The `allJobs.html.erb` view is a concrete example for this procedure.
It is also possible to define custom methods for fired actions in a view (e.g. pressing

the submit button), that are not associated with a view by themselves. The `submitJob` method for instance is only controlling but has no own view associated with it. This method only collects the input parameters from a view, validates them, saves them in the database, executes the follow up programs and redirects the user to the waiting view.

1.4 Models and the database

Models in rails are objects first of all. The difference to "ordinary" objects lies in the fact, that Rails models are directly associated to the database tables. In case of the model *raxml* exists a table *raxmls* in the database *RAxMLWS_development* that represents the model on database level. The columns contain the attributes and their values, where each row represents one instance of the model. When accessing and manipulating an instance of a model it is possible to manipulate the associated table entries without using SQL queries within the code. The concrete case of this rails application, there are only two models present. The *raxml* model (`raxml.rb`) that holds the submission parameters for the EPA and performs the validation of those. It also builds a shell script to submit the job to the grid. The second model *userinfo* (`userinfo.rb`) holds the user informations like IP and email address for statistical and job mapping purposes. The detailed table infos for the two models are displayed in table 1 and table 2. Observe that there is no declaration of attributes in the ruby scripts belonging to the model. The model knows its attributes solely from the table definition within the database.
The database-rails-interaction is basically configured by two different files. On the one side there is the connection file `config/database.yml` which contains the connection parameters to the database. On the other side there is the schema file `db/schema.rb` which holds the table definitions.

1.5 The scripts and programs

In this section I will give an overview with short descriptions about every script and program used by the web-application. The file endings are left out.

Table 1: raxmls table

raxmls attributes	**description**
id	The default primary key
query	The alignment type
alifile	Path of the alignment file on the hard drive
treefile	Path of the treefile on the hard drive
outfile	Obsolete relict
substmodel	The substitution model used e.g. GTRGAMMA
speed	`T` or `F` for using a faster algorithm
heuristic	`MP` or `ML`
h_value	Heuristic value
email	User email address
errorfile	Path of the SGE error file on the hard drive
parfile	Path of the partition-file on the hard drive
queryfile	Path of the queryreads-file on the hard drive
jobid	The unique jobid
user_ip	User IP address
job_description	User specific job description
b_random_seed	Random Seed for bootstrapping
b_runs	Number of bootstrap replicates
use_heuristics	`T` or `F` if heuristics is selected
use_bootstrap	`T` or `F` if bootstrapping is selected
use_queryfile	`T` or `F` if additional query-reads have been uploaded
use_clustering	`T` or `F` if query reads should be clustered
status	Job status running or done
created_at	Creation date
updated_at	Date of last update
mga	`T` or `F` if multigene alignment is used
use_papara	`T` or `F` if Papara instead of hmmer should be used

Table 2: userinfos table

userinfos attributes	**description**
id	The default primary key
ip	User ip address
saved_submissions	Obsolete relict
overall_submissions	Stores how many submissions were done by an IP address
created_at	Creation date
updated_at	Date of last update

1.5.1 app/views/raxml/

allJobs	This page is shown after the user has entered his email address in the look up jobs section. This form shows a table containing older submitted jobs allowing the user to select or delete them. The necessary job details like description etc. are provided by the corresponding controller function.
confirmation	A short confirmation message page for the user after he has submitted a message with the contact form.
contact	Contains the interface of the contact form.
index	This is the introduction page with some words about the EPA and the webserver.
look	The main view of the *look up jobs section*. A user can enter his job id or his email address to get access to his submitted jobs.
results	This view shows the results of the EPA job. Files are provided for download and the treeviewer applet is integrated showing the placement results at the bottom of the page.
submit_multi_gene	This view provides the user interface for the EPA. The user can upload his input files and enter his input parameters for a multi gene alignment submission.
submit_single_gene	Same as before, only for single gene alignment submissions.
treehelp	This help page contains some helpful information on the treeviewer, which can be selected in the results section.
wait	After a job is submitted, the user is redirected to this page as long as the submitted job is not finished. It also contains some useful information for the user like the job id.

1.5.2 app/controllers/

raxml_comtroller	This script is the heart of the whole application. It controls all the functionalities of the views and connects them to the models and the database respectively. Therefore there is a method for every view present along with some helper functions.

1.5.3 app/models/

fasta_to_phylip	This object expects a fasta formated alignment in an array and returns an phylip formatted alignment as an array. It is used to convert the uploaded fasta formatted alignment in raxml.rb
qstat_file_parser	Used by the raxml_controller getInfo method to parse the frequently generated qstat log file to show the server's capacity utilisation.
raxml_alignmentfile_parser	Used by the input validation method in raxml.rb to check if the uploaded alignment file is valid.
raxml_partitionfile_parser	Validates the partitionfile format.
raxml_queryfile_parser	Checks if the uploaded queryfile is in FASTA format.
raxml	The main model which holds and validates the submission parameters and files. It saves the input files on disk in the public/jobs folder and submits the job to the Grid. It also provides a method for the contact form to send the messages to the web-administrators and the hack for Simon to access the parallel version of the EPA through the description field.
raxml_results_parser	Collects all the files that are made available for download in the results view.
raxml_treefile_parser	This object checks if the format of the uploaded tree is in a valid newick format using a java program written by Simon Berger
userinfo	No additional methods were needed for this model, for this reason it is almost empty. Only the default methods from ActiveRecord::Base are used in the web-application for this model.

1.5.4 bioprogs/ruby/

`delete_old_jobs`	This program checks if the jobs saved in the database are older than 2 weeks. If this is the case, they are deleted. This program has been scheduled with *crontab* to run everyday at 4 am.
`fasta_file_parser`	A FASTA file parser used by raxml_and_send_email.rb to extract the sequences in case of a multi-gene-alignment from the query file.
`phylip_file_parser`	A PHYLIP file parser used by raxml_and_send_email.rb to extract the sequences in case of a multi-gene-alignment from the gene alignmentfiles.
`phyloxml_parser`	This object is used by the testing environment to check the correctness of the generated file.
`raxml_and_send_email`	The main script which is submitted to the grid. It processes the input parameters and starts the necessary programs for the job. It also sends the notification email to the user when the job is finished.
`reformat`	This is a helper script that can reformat some alignment formats into another. At this point, it can reformat PHYLIP formated alignments into STOCKHOLM format and FASTA and STOCKHOLM into PHYLIP format. It can also extract the cluster representative sequences from the uclust output files. This script is used by raxml_and_send_email.rb for this purposes.

1.5.5 bioprogs/java/

`convertToPhyloXML`	This Java program converts the EPA Classification files which contain the placements within the reference tree into a tree in PhyloXML format which can be read by the Archaeopteryx treeviewer. It also calculates the EDPL scores and also the experimental new weighted bipartition score and saves them within the PhyloXML file.
`treecheck`	This is the Java program used by the tree parser from above which validates the correctness of an input tree.
`treeMergeLengthsLabels`	A Java program that creates a correctly labelled tree that can be used by the convertToPhyloXML.jar program to build the PhyloXML formatted tree.

1.5.6 bioprogs/*other*

archaeopteryx	The treeviewer used to show the resulting placements. The corresponding applet lies in the public/ folder.
hmmer	The Hmmer tool. In this web-application only hmmbuild and hmmalign are used.
papara_ts	An alternative DNA aligning tool written by Simon Berger
raxml	The EPA.
raxml_parallel	The parallel version of the EPA which can be accessed by keywords within the description field of the submission form.
swps3	Swiss-Waterman algorithm used by the multi-gene-alignment pipeline.
uclust	This program is used if the user wants to cluster the uploaded reads.

1.6 The work-flows

In this section I will describe what steps are taken within the application after a single-gene-submission and a multi-gene-submission is processed. This might help in the future when debugging or extending of the web-application is needed. One examples for each case is described in the following.

1.6.1 EPA Single Gene Submission, queryreads uploaded, clustering activated

1. The user enters the webserver and sees the About EPA page.

2. He clicks on the Submit job button in the navigation to get on the submit page for single-gene-alignments.

3. When clicking on the button, the controller method *submit_single_gene* is called. It updates at first the box at the left which shows the server's capacity utilisation and initializes some input parameters of the submission page subsequent to that (e.g. selectable rate heterogeneity models).

4. The user enters his submission parameters. Checks "upload unaligned reads", selects the file with the reads and also checks "Cluster reads".

5. He presses the submit button and the *submitJob* method of the controller is called.

6. *raxml_controller.rb::submitJob* receives the input parameters, checks if the job is a single or multi-gene submission, builds the job directory, validates the input parameters (validate methods of the raxml model, in *raxml.rb*), saves the user data, calls the *execute* method of the raxml model and redirects the user to the wait view.

7. Within the *execute* method of the model, a shell file for the SGE is built which calls the raxml_and_send_email.rb script with the input parameters and an *echo*

done! command at the end. This script is then submitted to the Grid and the method returns.

8. raxml_and_send_email.rb parses the input options first, calls uclust to cluster the reads, gets the cluster representative sequences and aligns them with hmmer to the input alignment. After that, RAxML is called. When RAxML is finished, the right labelled tree is created (*treeMergeLengthsLabels.jar*) and the PhyloXML file built. Subsequent to that, an email is sent to the user (if he has entered his email address) containing a link to his results. As a last step, the server status is refreshed by updating the *qstat.log*.

9. *We assume now, that the user lets the wait view open and that he waits until the job is finished*

10. The wait view refreshes every few seconds. Therefore the wait method within the controller is called every time. It checks within the *submit.sh.**, which are the output log files of the SGE, if a "done" line is present. The user is redirected to the results view if this is the case.

11. After the submission is finished, the user is redirected to the results view, calling the results method of the controller which loads all the files that can be downloaded from the page. It also determines and delivers the right cites for the submission and gives the view other parameters for e.g. the treeviewer applet to run.

1.6.2 EPA Multi Gene Submission, queryreads uploaded, clustering activated

1. The user enters the webserver and sees the About EPA page.

2. He clicks on the Submit job button in the navigation to get on the submit page for single-gene-alignments.

3. When clicking on the button, the controller method *submit_single_gene* is called. It updates at first the box at the left which shows the server's capacity utilisation and initializes some input parameters of the submission page subsequent to that (e.g. selectable rate heterogeneity models).

4. The user clicks on the *Multi Gene* button. The *submit_multi_gene* method of the controller is called. It also updates at first the box at the left which shows the server's capacity utilisation and initializes some input parameters of the submission page subsequent to that (e.g. selectable rate heterogeneity models).

5. The user enters his submission parameters. Selects the file with the query reads and also checks "Cluster reads" checkbox.

6. He presses the submit button and the *submitJob* method of the controller is called.

7. *raxml_controller.rb::submitJob* receives the input parameters, checks if the job is a single or multi-gene submission, builds the job directory, validates the input parameters (validate methods of the raxml model, in *raxml.rb*), saves the user data, calls the *execute* method of the raxml model and redirects the user to the wait view.

8. Within the *execute* method of the model, a shell file for the SGE is built which calls the raxml_and_send_email.rb script with the input parameters and an *echo done!* command at the end. This script is then submitted to the Grid and the method returns.

9. raxml_and_send_email.rb parses the input options first, calls uclust to cluster the reads and gets the cluster representative sequences. To process the multi gene alignment with RAxML some more steps have to be made. At first the multi gene alignment is split in single gene alignments using RAxML in combination with the uploaded partitionfile. After that for each single gene alignment, the sequences are disaligned and an all vs all of alignment sequences against the query reads with SWPS3 is performed and the maximum scores are saved for each read. The read is assigned to the top scoring gene. After all reads have been assigned to a gene, they are aligned with hmmer to the corresponding single gene alignments. RAxML is then performed with every resulting single gene alignment. When RAxML is finished, the resulting classification files are concatenated and the right labelled tree is created (*treeMergeLengthsLabels.jar*). Subsequent to that the PhyloXML file built (*convertToPhyloXML.jar*) and an email is sent to the user (if he has entered his email address) containing a link to his results. As a last step, the server status is refreshed by updating the *qstat.log*.

10. *We assume now, that the user lets the wait view open and that he waits until the job is finished*

11. The wait view refreshes every few seconds. Therefore the wait method within the controller is called every time. It checks within the *submit.sh.**, which are the output log files of the SGE, if a "done" line is present. The user is redirected to the results view if this is the case.

12. After the submission is finished, the user is redirected to the results view, calling the results method of the controller which loads all the files that can be downloaded from the page. It also determines and delivers the right cites for the submission, and gives the view other parameters for e.g. the treeviewer applet to run.

1.7 Testing

The RAxML webserver contains also a small test-suite that can be started by typing `rake` into the terminal when you are in the RAxMLWS folder. The tests contain mainly user cases where different input parameters are tested and also if false inputs lead to an expected error. At this moment, there are 24 different user inputs tested which are defined

in `test/fixtures/raxml.yml`. In `test/functionals/raxml_controller_test.rb` the actual test class is found. It tests some methods of the raxml controller using in the case of the submitJob method the fixtures mentioned above.

1.8 Miscellaneous

In this section I will give some miscellaneous notes on some aspects of the Rails application.

1.8.1 Configurations

All of the internal configurations of a Rails application are done within the *config* folder. For this webserver project only the *environment.rb*, the *database.yml* and the *mongrel_cluster.yml* if available are of importance.
In the *database.yml*, rails is configured how to access the database. This can be done for the development, test and production environment. In our case the development and production access are the same, because the web-application is/was not running on the same machine in production and development mode at the same time, and for that reason I decided that it was not necessary to differentiate.
In the *environments.rb* class it is possible to define in which mode the web-application should run (development/production). It is also possible to define Rails internal variables in our case saved within the *ENV* hash.
The last file configures the mongrel cluster where to run, where log is written to, the starting port, in which mode, where the pid file is saved and at last how large the cluster is supposed to be.

1.8.2 Access Rails environment in bioprogs folder

Someone who is new to Rails might wonder how the scripts lying in bioprogs folder which has been artificially placed within the Rails folder system access the Rails internal variables. Unfortunately some lines of code had to be added to the scripts that make use some Rails internal variables. This can be seen in header of the *raxml_and_send_email.rb* script. At first we define the RAILS_ROOT environment by hand using a ruby method and then we load the *environment.rb* from the config folder.

1. `RAILS_ROOT = File.expand_path(File.join(File.dirname(__FILE__), '../..'))`

2. `require "#{RAILS_ROOT}/config/environment.rb"`

1.8.3 Scheduling tasks

As in case of *delete_old_jobs.rb* it is possible to schedule tasks. This is done by entering the following command in the terminal and editing the following file:
`crontab -e`

What can be seen is for example:
```
# m h dom mon dow command
0 4 * * * /home/denis/work/RAxMLWS/bioprogs/ruby/delete_old_jobs.rb > some.log
```
This means, that the script is executed at 0 minutes at hour 4 any day, any month and any year. Or in other words, everyday at 4 am. If you want to execute the program as root, just open crontab as root or using sudo.

1.8.4 Accessing the parallel version of the EPA

It is possible to access the parallel version on the EPA through the description field in the submit forms. Just enter the following, where *number* can be any number:
`<?number?>`
This "code" is parsed in *raxml.rb::parseDescription*

2 Installing the web application on a new machine

In order to setup the EPA webserver, it is not enough to download the repository files alone. The repository misses some files especially some programs and configuration files that are more or less dependent on your local computer like e.g. the database. It is also necessary to prepare your system for the Rails application. I will try to give a step by step manual that should suffice for running the webserver on a new server or a local computer for development purposes. I will note if some steps are not necessary for the developmental purpose. Some steps might be obsolete, I just did not have the time to check which packages are really necessary for the application to run. During the development I tried out different things so I might have some obsolete relics installed.

2.1 Ubuntu Packages

1. install ruby: `sudo apt-get install ruby-full`
2. install rails: `sudo apt-get install rails`
3. install rubygems: `sudo apt-get install rubygems`
4. install mysql: `sudo apt-get install mysql-client mysql-server`
5. install Sun Grid Engine: `sudo apt-get install gridengine-client gridengine-exec gridengine-qmon gridengine-master` Tell Ubuntu to configure SGE automatically, SGE-CELL: default, SGE-Master-Host: *your_computer_name*. Run qstat afterwards and check if it exits without showing any error messages. If not, everything should be in order. It happened once to me that the sge_master refuses to start or that he does not recognize *your_computer_name*. To fix the first issue you have to replace the openjdk java jre trough the sun-java6-jre. I cannot give a tutorial here how to do that, because as I found out it strongly depends on the

ubuntu version you use. But there is plenty of help available on the web. I fixed the second issue by editing */etc/hosts* and assigning *your_computer_name* the same IP as the localhost (127.0.0.1), restart sge_qmaster by killing it first and starting it afterwards with sudo rights. You can also reboot if the restart of the qmaster did not solve the problem.

6. install mongrel: `sudo apt-get install mongrel`
7. install mongrel cluster: `sudo apt-get install mongrel-cluster` (not needed for development (not dev.))
8. install apache: `sudo apt-get install apache2` (not dev.)

2.2 Setting up Rails

1. change to the directory where you want to install the web-application
2. create Rails app: `rails RAxMLWS` it should have created a folder with the name RAxMLWS in that folder along with some child directories and files.
3. change in the RAxMLWS directory
4. we will now install some gems. Some of them might be obsolete like installing the mongrel gem, but anyway, I will do so now. Install the following gems with sudo rights with the following command:
   `sudo gem install` *`package`*
   The packages are:
   - cgi_multipart_eof_fix
   - daemons
   - fastthread
   - gem_plugin
   - highline
   - i18n
   - mongrel
   - mongrel_cluster
   - mysql (Note! Almost in every installation I made, the mysql gem could not be installed out of the box because it misses some libraries. If you get an error, check if you have installed *libmysqlclient-dev* and *libmysqlclient16-dev* in your system)
   - open4
   - Platform
   - popen4

- responders

5. download the RAxMLWS repository and place all the content in the RAxMLWS folder you created beforehand. Override every folder and every file that has duplicates.

6. type `mysql -u root -p` into the terminal and enter your root password for mysql that you specified during the installation. You will end up with a MySQL command-line. If you like you can create a new user now that has some limited access to the database (the application can use this user for safety reasons). However, I did not do that. Create a development, test and production database : `CREATE DATABASE some_name;`. Exit the MySQL command-line after that.

7. change into the `config` folder and open the database.yml. Edit the file such that it fits your needs, specifying as which user the webserver should access the database, the password and which database should be accessed in case of development, test or production mode. Also change *adapter* to mysql.

8. get back to the command-line, change one folder up and call `rake db:migrate` for creating all the tables needed by the webserver.

9. open `environment.rb` and add the following lines on top to it

   ```
   ENV['MAILSERVICE_ADDRESS'] = 'localhost'
   ENV['SERVER_IP'] = "your external IP unless" defined?  SERVER_IP
   ENV['SERVER_ADDR'] = "your external address with DNS name" unless defined?  SERVER_ADDR
   ENV['SERVER_NAME'] = "only the DNS name" unless defined?  SERVER_NAME
   ENV['MAIL_SENDER'] = 'localhost'
   ```

   In case of my developing machine it looked like this:

   ```
   ENV['MAILSERVICE_ADDRESS'] = 'localhost'
   ENV['SERVER_IP'] = '131.159.28.62' unless defined?  SERVER_IP
   ENV['SERVER_ADDR'] = 'http://lxexelixis1.informatik.tu-muenchen.de' unless defined?  SERVER_ADDR
   ENV['SERVER_NAME'] = 'lxexelixis1.informatik.tu-muenchen.de' unless defined?  SERVER_NAME
   ENV['MAIL_SENDER'] = 'localhost'
   ```

   If the java applet does not find any files, you probably forgot to specify the port for `ENV['SERVER_ADDR']`. In case of default webrick this would be :3000.

10. create the folder `public/jobs`

11. create the folder `tmp/files` and within this folder a file named `qstat.log`

2.3 Gridengine

1. check now if sge_qmaster and sge_execd are running by typing for example `ps -aux | grep sge` in the terminal. If they are not running, start them with sudo rights.

2. get back on the command-line, type `qmon` to open the configuration GUI of the gridengine (Note! Lately qmon reports an error because it misses some fonts. If you get this kind of error, just install the packages *xfonts-100dpi* and *xfonts-75dpi* and restart).

3. when the GUI is open, click on the *Host Configuration* button. Within the tab *Administration Host* you should see *localhost* and *your_computer_name*, if not add the missing. Under the tab *Execution Host* the hosts should be *global* and *your_computer_name*. Last but not least we need to add the submit host *your_computer_name* within the tab *Submit Host*.
   Exit the Host Configuration and proceed to the button *Queue Control*. We will add a new queue now. Click on *add* on the right. A new panel opens. Enter a queue name at the top and enter *your_computer_name* in the field below that and push the red arrow button left to it. Below that, the tab *General Configuration* should be open. Change the Shell field to the shell your system uses. As a last step assign the number of slots as needed in the slots field. Press *Ok* in the upper right after that. Exit qmon.
   You could write yourself a little shell testscript and submit it with `qsub` to the queue to test if everything is in order.

2.4 other programs

Check if the `bioprogs` folder looks like this:

```
archaeopteryx  hmmer  papara_ts  raxml_parallel  swps3
dummy          java   raxml      ruby            uclust
```

Every folder should contain the corresponding program. raxml, raxml_parallel and papara_ts have to be downloaded and compiled. Uclust should also be downloaded from the uclust site if not present (Note! The uclust creator sometimes changes his command-line formats with new versions, after downloading all the programs needed you should run the tests.)

2.5 Starting the Development-Server

1. change to the root directory of the rails application

2. enter `script/server webrick` , it shows you under which IP this little test webserver starts the web-application. Open firefox or another browser and enter this address and the port e.g. 127.0.0.1:3000. You should see the rails welcome page. Extend the address to 127.0.0.1:3000/raxml , you should see the About EPA page now. The EPA Webserver is now ready for development.

2.6 The Production Environment

The next steps are only necessary for the production server.

1. open `environment.rb` in the config folder and open it with an editor. Uncomment the line `ENV['RAILS_ENV'] ||= 'production'` and exit.

2. Now we configure the mongrel_cluster first. Change in the `config` folder of the Rails application, create the file `mongrel_cluster.yml` and open it with an editor. Edit the file such that it looks like this but with your parameters instead:
```
---
address: 127.0.0.1
log_file: log/mongrel.log
port: "8000"
cwd: /home/denis/work/RAxMLWS
environment: production
pid_file: tmp/pids/mongrel.pid
servers: 3
```

3. In the last step, we will configure the Apache now. This will be a basic configurations only to run the app. For more advanced configurations multiple tutorials are available on the web. First we need to load some modules. Change to the folder `/etc/apache2/mods-enabled` and display the content of this directory. These are the modules enabled at the moment. Here is my list of modules that are enabled compare these with yours and enable the missing ones by typing `sudo a2enmod` *module_name*

```
alias.conf            autoindex.load    negotiation.load
alias.load            deflate.conf      passenger.conf
auth_basic.load       deflate.load      passenger.load
authn_file.load       dir.conf          proxy_ajp.load
authz_default.load    dir.load          proxy_balancer.load
authz_groupfile.load  env.load          proxy.conf
authz_host.load       mime.conf         proxy_connect.load
authz_user.load       mime.load         proxy_ftp.load
autoindex.conf        negotiation.conf  proxy_http.load

proxy.load
proxy_scgi.load
reqtimeout.conf
reqtimeout.load
rewrite.load
setenvif.conf
setenvif.load
status.conf
status.load
```

4. Change now into the folder `sites-enabled` and open the 000-default file with an editor. Edit the file such that it looks like this but with your parameters:

```
<VirtualHost *:80>
ServerName http://hitssv106

DocumentRoot //home/denis/work/RAxMLWS/public
<Directory />
Options FollowSymLinks
AllowOverride None
</Directory>
<Directory /home/denis/work/RAxMLWS/public>
Options Indexes FollowSymLinks MultiViews
AllowOverride None
Order allow,deny
allow from all
</Directory>

<Proxy balancer://cluster/>
BalancerMember http://hitssv106:8000
BalancerMember http://hitssv106:8001
BalancerMember http://hitssv106:8002
</Proxy>
ProxyPass /images !
ProxyPass /stylesheets !
ProxyPass / balancer://cluster/
ProxyPassReverse / balancer://cluster/
ProxyPreserveHost On

<Proxy *>
Order deny,allow
Allow from all
</Proxy>
</VirtualHost>
```

   Some words about this file: Everything until the Proxy tag should be clear how this has to be fitted individually. The `BalancerMember` are the mongrel units from the mongrel cluster. We previously defined 3 Servers starting at port 8000 in the mongrel_cluster.yml. These three units are listed here, one on port 8000, one on port 8001 and one on port 8002. If you have made other configurations in your mongrel_cluster.yml you have to change this here corresponding to that.

5. The last thing we have to do now is to start the mongrel cluster and restart Apache. Change in the Rails application root folder and type:
   `sudo mongrel_rails cluster::start`
   after that type:

`sudo /etc/init.d/apache2 restart`
The EPA webserver should now run under the server's address.

3 Bugreport

3.1 SWPS3 Matrices, multi gene alignment pipeline

The DNA Matrix (`swps3/matrices/dna_matrix.mat`) and the Blosum62 (`swps3/matrices/blosum62.mat`) matrix for the SWPS3 algorithm are not well balanced. Amino Acid matches are favoured by multiple factors (in my example this factor reached 50). This way the SWPS3 might assign DNA sequences to the a wrong amino acid gene instead to the right DNA encoded gene. As a consequence, every read is assigned to the amino acid encoded genes resulting in really bad alignments generated by Hmmer. The DNA matrix weights have to be adapted.