**Test Forum Project**

**The Brief**

Implement simple forum/messageboard web application based on ruby on rails framework.

Application must run on linux (pick your favorite distribution) and use postgresql database.

Functionally, ideally application could use login, roles (admin, user) and access privileges - user can add/delete their own posts and only admin can delete other users posts. Simple flat (everyone can post) functionality is acceptable also.

Put some effort into UI design also, so it would not be just plain text.

Test the application. Use your best judgment on how to approach testing (not to specify, but as an example unit, functional UI, load) and pick any tools you prefer. What we'd expect are instructions on how to install the application, how run it, and in quite detailed summary on: testing strategy, tools, code, scripts, results.

**Design**

A forum web application was chosen, with the following features:

* Header
* Visible on all pages, even when scrolled passed
* Shows 'login' when not logged in
* Otherwise, shows specific user options
* Footer
* Basic. Does nothing but include links
* User Signup (for non-admin)
* Unique email verification
* Activation email
* Form validation
* User Login
* Activation check
* Session remember
* Cookies
* Forgot password
* Email token to reset
* Forum root
* Sign up/Login page
* Redirect to in the event of viewing pages while not being logged in
* Forum Home
* Paginated list of forums
* Expected to be fixed, only admin can edit/delete/create (via button link to form page)
* Latest updated topic shown for each
* Key text as links, e.g. forum topics page, latest topic, latest user
* Forum Topics
* Paginated list of topics for given forum
* Ordered by the latest post in each topic, newest first
* Admin can edit/delete/create (via button link to form page)
* Non-admin can only edit/delete own and create
* Last post info for each topic
* Key text as links
* Topic Posts
* Paginated list of posts for given topic
* Ordered, oldest first
* Admin can edit/delete/create (via quick form at bottom of page)
* Non-admin can only edit/delete own and create

The visual style of the site was chosen to be done using Bootstrap. It is widely in use and has plenty of documentation for someone learning to use CSS. Parts of Bootstrap used included:

* Affix (for the fixed header)
* Dropdown for User options in header
* Jumbotron (for the homepage)
* Buttons
* Forms + Inputs
* Grid (for layout)
* Tables (striped, for displaying content)
* Modals (for editing topic/forum titles and post content)
* Alerts
* Breadcrumb (for page links)
* Navbars
* Tooltip (for seeing full content, when truncated)

DB schema design:

* Forum
* has many topics
* has many posts, through topics
* deleting a forum also deletes topics and posts
* Topic
* has one forum
* has many posts (at least one)
* has one last post (for ordering purposes)
* deleting a topic also deletes posts
* Post
* has one topic
* has one user

For users to be able to delete their own topics, it was necessary to always associate the ownership with the opening topic post's ownership. Therefore, a user can only delete a topic where the opening post belongs to them. It was decided that deleting the first post also should delete the topic as well.

**Installation Instructions**

During development, Ubuntu 14.04 LTS was used. The following instructions assume that you are using a similar OS and starting from scratch:

Install RVM (Ruby Version Manager):

* instructions here: <https://rvm.io/>
* if the 'gpg –keyserver' command fails, it may be necessary to use the following command instead: 'curl -sSL https://rvm.io/mpapis.asc | gpg --import -'

Install Ruby and Rails:

* 'rvm install 2.2.1' (for Ruby)
* 'gem install rails -v 4.2.2'

Install and configure Postgresql:

* 'sudo apt-get install postgresql-9.3 postgresql-contrib-9.3 libpq-dev'
* 'sudo -u postgres createuser -s pguser'
* 'sudo -u postgres psql'
* create a password for pguser: 'alter user pguser with password 'password';'

Install and configure Git:

* 'sudo apt-get install git'
* 'git config --global user.name "Your Name"'
* 'git config --global user.email "Your Email"'
* 'git config --global push.default matching'

Get the source code:

* 'git clone <https://github.com/andrewmurray21/test-forum> test-forum'

Setting up the environment (within test-forum folder):

* install gems: 'bundle install –without production'
* create DBs: 'bundle exec rake db:create'
* add migrations: 'bundle exec rake db:migrate'
* add test data: 'bundle exec rake db:seed'

Start the server

* 'rails server'

If you wish to log in with the admin user, you can use the following credentials:

* Email: '[testforum63@gmail.com](mailto:testforum63@gmail.com)'
* Password: 'password'

**Testing Strategy**

The following tools were used in testing the application:

* Minitest
* guard-minitest for quick feedback after saving code changes
* minitest-reporters for better view of instant results
* byebug/web-console for easy viewing of requests/params on the page
* mini\_backtrace for easier debugging
* simplecov for code coverage
* Faker to generate more realistic seed data
* Jmeter for load testing

Utilising Rails' different environments, automated testing was done using the test environment, while the development environment was used primarily for checking visuals.

DB seed and fixtures were used to create ample testing data by populating the database.

Unit tests for all Models were written early in the development life cycle. Using assertions, they covered at least one check for every validation and for every method. Code coverage for models is 100%

Controller actions were more functional so required both unit test-style tests and functional ones. The intent was to follow a TDD approach but it quickly became apparent, since the project was a learning exercise and a lot in the functionality was unknown, that this approach wouldn't be feasible. Tests were created after development covering every path through each method. The key things tested for were request, parameters, redirects and flash messages. A user permissions text matrix was created to ensure that nothing was missed:



Note: “Other” user assumes that in the case of Topic/Post edit and delete, they are not the owner.

Also using the matrix above, Integration tests were created concerning interoperability of controllers and functional testing, including presence tests for View components of the application (dependent on particular permission levels). A lot of the testing here negated the need for a point-and-click testing app like Selenium, but some time was invested in learning Capybara. Code coverage is currently 100% (viewable at /coverage/index.html).

It was decided against using Minitest:Benchmark for performance profiling of the application simply due to time constraints. Jmeter was used instead, which is an open source, free-to-use GUI application which can simulate load on an application. It even provides recording functionality, making it very simple to pick up and use. Again, due to time contraints, especially when trying to run a significantly tweeked and lenghty load test, only a simple example was included in the testing (stored in git /project/jmeter). The results of which are below:

10 concurrent users, following a simple path through the application 10 times (Root > Login > Forum Home > Topics > Posts > User):



As can be seen from the example above, no errors occurred during the test run. When this same test was run during implementation, the Average column figures were significantly higher. Based on viewing the page requests in the server log, it was evident that this was due to a large number of calls to the database. After correctly updating the controller variables to .includes(:xyz), the figure decreased satisfactorily, reducing the n+1 queries.

The point of failure of the application is somewhere close to 100 consecutive users, as can be seen from the table below.



The above tables are just examples of what could be done by load testing of the application. Preferably though, each page should be tested individually, or in a shorter scenario. The scenario above could be used to provide load to the application while assessing the smaller scenarios at the same time.

As well as the simple routes covered in the test above, the important test cases should be database intensive activities, such as creating new users/forums/topics/posts and editing/deleting each of them. One of particular interest is deleting a forum which contains many topics and posts.

**Project Status and Further Improvements**

As mentioned in the previous section, despite 100% code coverage, there are a number of tests which could be added to consider the project complete. These include creating functional UI tests, perhaps using Capybara, if necessary, and increasing the number of load tests ran through Jmeter.

As well as increasing the testing, there were also a number of other features that could have been added, had time allowed. Such as:

* Forum up/down arrows – the intention was that the Forum Home remain fairly static but with the addition of a new forum, the administrator could move the new one higher in the page. This could be implemented using a priority field in the forums table of the database.
* Search – should display a new page based on results of search for post content, topic titles and user names.
* Uploaded avatars, instead of gravatars – The latter was used to reduce space taken on the production server but could easily be implemented using the gems, carrierwave, mini-magick and fog. These would allow easy upload and resizing.
* Image upload within posts – same as above