**Doctor Self Study**

**Design Document:**

This application uses databases to organize and store information about Users, Questions, Assignments, and Administrators in a way that allows the application to know when to send out links to a particular user to answer a particular question. It uses a CRUD (Create Read Update Delete) system to manage these items while providing back-end algorithms to associate questions with the appropriate users and mail out links for the users to respond.

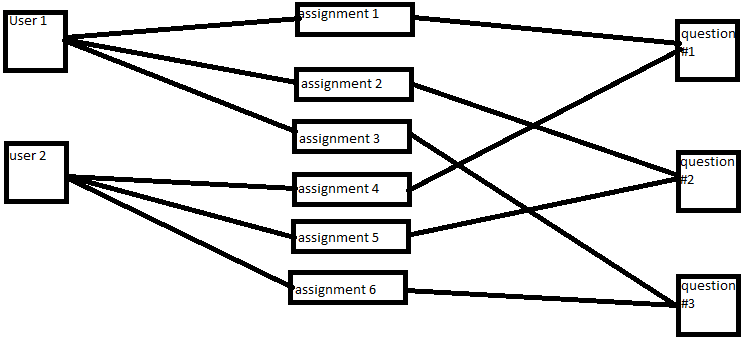
**Basic Rails application structure:**

The application is structured using the Ruby on Rails MVC architecture and leveraging the powerful Rails Active Record class for seamless automated database interaction. Active Records play the role of a Model in the MVC architecture and define the behavior of each Active Record. They define methods for manipulating the state of these objects. The controllers inherit from the Rails Application controller class and control how the application responds to specific requests in the form of URLs. Each Controller defines how an Active Record responds to such requests and sets up the state so that the View can display properly. An example of an Action in a URL would be Questions/index. This refers to the index action in the Questions controller. The controller would in this case set up an instance variable @questions, which would contain a list of all the instances of the ActiveRecord Question stored in the Questions table in the database. This instance variable would be referenced in the index view. Views in Ruby on rails are structured as embedded Ruby files that generate HTML and represent the response to a request. These are for the most part HTML files but with the ability to embed ruby code and algorithmically generate HTML. A typical request is processed using the Routing file Routes.rb, which matches the request to the correct controller and action. The action in the controller uses Active Record methods to set up state variables for the View to use, and responds with the proper view, whose embedded ruby is resolved to HTML and which processes using the state set up in the controller. I strongly recommend the Lynda.com Ruby on Rails 3 essential training tutorial for anyone interested in learning this framework.

**Application pieces:**

Our application maintains 4 Active record objects: Admins, Users, Questions, and Assignments. Users represent the residents that are responding to questions sent by the doctors. Questions represent the individual questions which are assigned to groups of users. Admins represent the Doctors who are managing the website and creating the questions. Finally, Assignments represent a User Question pair, that is a unique link between a user and a question and keeps track of that user’s performance and history specific to that question.

Example system with 2 users and 3 questions. In this example each question is assigned to all users.



Users are classified by their year and rotation, representing what point they are at in their residency. The doctors categorize the questions they create by which rotation and user they want to assign the question to. In both the user class and model class, there are methods for creating assignments based on a user’s rotation and year. These are implemented as call backs. Rails call backs allow the developer to specify methods that should be called during certain points in the lifecycle of an active record instance. In this case the methods that create assignments are run after a record is created and every time it saves. Thus, when a user or question is created, the application will automatically generate an assignment between the appropriate users and question, and whenever a user or question is updated, the application will re-assign the questions in case the rotation or year parameters have changed while preserving assignments that have been answered in the past.

**Access and Authentication**

In addition to the 4 active records and their respective controllers and view files, we also have an Access controller, which contains the functionality for user and admin authentication and logging in and out, as well as a main menu screen, which links to the action in the different controllers. There is no Active Record Model associated with the access controller. Instead, the Access controller contains a few utility forms and menu actions for logging in logging out and displaying the main menu.

Authentication in our application is slightly complicated. Passwords are entered plain text when creating a user or Admin user. We use a callback that will detect the presence of this password parameter, a value that is only persistent in the model and not in the database and follow the following steps to encrypt and save:

1. Use a combination of the username and some random text to generate salt, a random and unique set of characters that we add to the user’s password to make the encryption process more secure.
2. Encrypt the salt for even more randomization and save the salt to the database.
3. Combine the salt and the password and encrypt that phrase and store that hashed phrase in the database.

In order to authenticate during login, the model uses the saved salt with the password the user entered and encrypts it using the same algorithm described above, and then compares the resulting hash with the hashed password stored in the database. If successful the authenticate method will return a user or admin user object, and will write the user id and admin status to the session file. Subsequent controllers use these parameters in the session file to confirm whether the user is logged in and redirects to the login page if not. Some pages require an admin user to be logged in while others can be accessed by users. The session file keeps track of boolean that determines whether the user is an admin to decide whether the user has permissions to view the page.

**Scheduling and Resque Scheduler:**

Scheduling refers to the way in which questions are pushed to the residents. Each question object has a schedule item stored in the database which specifies the time at which it should become available to the user. Scheduling is implemented using Resque scheduler, an open source ruby gem which is build off of the open source gem Resque, which in turn depends on a paid service called redis. Thankfully the Resque functionality can function at the level of the free (nano) tier and requires no payments.

Redis is an object database service on which Resque relies on to store Ruby objects while they are waiting to be performed (as opposed to running immediately on the server). Resque operates on the basis of jobs, which are small ruby classes that define a single action to be performed. Typically these represent a background job like sending an email (in our case) or uploading files or updating some table. Resque organizes these jobs into ‘queues’ on the Redis server and a resque process (a “worker:” rake Resque:work from command line) running in the background on the server will process the job as soon as it can. Resque Scheduler builds on this functionality by using multiple queues. In addition to using a queue from which the worker pulls jobs to perform, Resque scheduler maintains a delayed queue, which stores jobs that are scheduled to be run eventually, but which should not be executed now. This, finally, is where our push jobs are stored. Whenever an assignment is created, a callback calls the resque Scheduler method enqueue\_at(timestamp, jobClass, parameters). This puts a job into the delayed queue which when performed, will send a link to a resident which will take him to the website to record his answer to the question. The server must run yet another background process to poll the delayed queue and determine which if any jobs need to be moved over to active queues. This scheduler process is run from the command line with rake resque:scheduler. It is important to note that both of these processes (rake resque:scheduler and rake resque:work) must be running for the system to function.

Two other tutorials for configuring [Resque and redis](http://blog.togo.io/news/resque-with-redis-to-go/) and [Resque Scheduler and redis](http://blog.togo.io/how-to/resque-scheduler/)

As a caveat to this whole discussion, this is how the application is intended to work, but there are bugs which unexplainably stop this from actually happening. See this Stack Overflow post for a description of the problem: [Stack Overflow Resque Scheduler error](http://stackoverflow.com/questions/15858776/mysql-error-using-resque-with-rails-table-does-not-exist)

In addition I’ve opened an issue describing the problem on the Resque Scheduler github page [here](https://github.com/bvandenbos/resque-scheduler/issues/215)

Finally, since our Doctors are working on a budget, it is important to note that the use of this Resque scheduler application requires background processes that if run all the time would cost a lot to run (about $70 per month on Heroku’s service, perhaps less on production servers). We chose Heroku because it has free hosting services, but in order to run these jobs, an open source module called Heroku Autoscaler is reuired. This module automatically scales up the server to devote processing power toward the Worker and Scheduler processes when they are run and then scales them down when they complete. Using this autoscaler it is possible to use Resque Scheduler with Heroku for free. Importantly though, to implement this all Active Record Models and Job classes must extend the module. See [this](http://verboselogging.com/2010/07/30/auto-scale-your-resque-workers-on-heroku) and [this](http://blog.togo.io/news/resque-with-redis-to-go/) page for more details on the Autoscaler.

There will doubtless be some issues fully integrating Resque scheduler into the Heroku app but because we could never get Resque running, we were unable to fully flesh out the isues.

**Layouts**

We use Twitter’s bootstrap layouts as a tool for styling our website. Refer to Twitters Documentation about how to integrate their layouts into further extensions of the app. One thing to note about layouts is that all assets in the assets folder need to be “precompiled” before uploading to heroku or Heroku will not work AT ALL**. You must run “rake assets:precompile”** to do this and then make sure to commit the resulting file to the heroku repository. Note the precompile command will take a while (~5 min) to run as the process is quite slow.

**Notable code:**

The following files are primarily associated with Resque Scheduler and are prime candidates for the bug that is making the scheduling system not work.   
1 App/jobs/scheduler\_job.rb  
2 config/initializers/resque.rb  
3 lib/tasks/resque.rake

The heroku autoscaler module lives here: lib/heroku\_resque\_auto\_scale.rb  
You shouldn’t need to edit this module unless something changes. . .

A Mail interceptor that catches all outgoing mail and redirects it to a hardcoded email (so you don’t spam users or fake email addresses) lives here: lib/MailInterceptor  
YOU SHOULD CHANGE THE HARDCODED EMAIL IN THIS FILE. It is currently set to send the email to my (will nance’s) personal email address. I do not want these messages and I’m sure you will in order to test that it works.

Furthermore, config/initializers/setup\_mail.rb contains the username and password information associated with the website’s gmail address. Sending mail from gmail is free so we set up an account for the application. The is code at the bottom to register the mail interceptor with the app. Notice at the end I’ve commented out a conditional that limits the interceptor to work only in development mode. You’ll want to uncomment this line to send emails correctly in production mode when you are sure that you’ve fixed the bugs with the scheduler.

**Creating a temporary Admin Account**

When you first initialize the website the list of admins will be empty, thus you will not be able to enter the website because there won’t be any login information. To fix this enter the rails console (command: rails console , or if on heroku: heroku run rails console) and enter the following command to get access to the site:

admin = Admin.create( :firstName => "john", :username => "adminadmin", :lastName => "doe ", :email => "nobody@nowhere.com" , :password => "password")

This will create an admin account with the username adminadmin and password password. You can then log in with this username, create the appropriate admin accounts from the GUI and then delete the temporary admin account.

**RedisToGo account**

This is the supporting account that maintains the redis queues used by resque. The url for the database is hardcoded in **config/initializers/resque.rb**   
as well as the email and password (wllnance ‘at’ gmail ‘dot’ com and doctorselfstudy respectively). **These should be changed** to the developer’s email when you take over this project. Editing the user account info online is trivial here: <https://redistogo.com/account/edit?language=en>

**Doctor Self Study gmail account**:

Username : Doctorselfstudy  
Password: dsspassword

**Troubleshooting**

First of all, using a mac or Linux is a million times easier than using windows with Ruby on rails. I would Strongly recommend using a \*nix system to develop on. As a warning you should know that the code in the current state is not completely platform agnostic. There are some errors that persist on Linux machines that we suspect have to do with case sensitive vs non-case sensitive systems when switching to a linux or mac I believe there may be errors in the users controller and don’t guarantee there aren’t other OS bugs because we developed on windows. However, I would argue that the stability from using a Unix setup outweighs these problems. If you get strange SQL errors where a field you know should be there doesn’t exist, there is likely a camel-cased variable or database column name that is causing the problem. Refactoring variables and column names to use under\_score names versus camelCase will probably resolve most of these problems. Also note that Heroku and most production servers are Linux-based so developing in a Unix environment will save you some surprises when deploying. That said here are some troubleshooting tips.

Netbeans says server is already running? Delete files in tmp/pid/ and try again

If you get an error relating to SSL errors and persistence during bundle install, try following the steps here: <https://gist.github.com/fnichol/867550>

If you have an issue with running the app where it can’t find a gem which you know is installed, try running this from command line in the application directory (note there are two dashes): bundle install –path vendor/cache

Error relating to coffee scripts on windows 8? Need to install therubyracer but can’t? Try this: <https://github.com/hiranpeiris/therubyracer_for_windows>

Still hate your life and can’t install therubyracer? So did we, but this will point you in the right direction: <http://stackoverflow.com/questions/6356450/therubyracer-gem-on-windows>

Still hate your life and can’t get it to work on window? See above where I say that you shouldn’t use Windows to develop for this project (or Ruby on Rails in general). Try this: [Ubuntu Wubi installer](http://www.ubuntu.com/download/desktop/windows-installer)