SE31520 ASSIGNMENT 1

Part 2: Forum for the CSA

A report on the CSA Application

Jake McDonagh
jam93@aber.ac.uk

Contents

Introduction

The following report on this assignment will contain information about the overall architecture of the CSA Application which includes both the MVC design pattern used within the application along with other key files that are also important to application as a whole. As well as this, the report will also consist of documentation for both the RESTful client along with a section where I will be discussing how successful the testing using the provided Cucumber feature file went. This report will also have a flair section where I will be detailing the extra work and features that were created outside of the initial requirements. Finally, I will evaluate all of the work that I have created for the purpose of this assignment and discuss my overall thoughts on this project which includes what parts I found easy or hard as well as the parts which I have personally omitted from doing within the assignment. I will also be critiquing the appropriateness of my design and the methods that I had used in order to complete the following tasks.

Troubleshooting

Once the source code has been unzipped, the CSA application should work without any problems. However, if there are problems with running the CSA applications, please take the following steps to ensure that everything is correctly set up:

1. Open up your personal terminal of choice and navigate your way to the CSA folder.
2. Run the following command: rails db:drop db:migrate db:seed
3. By doing this, you are effectively dropping the entire database and are running the migration files to create the database from scratch again. The consequence of doing this however is that you also drop the test environment for the cucumber feature file.
4. Run the following command: rails db:seed RAILS_ENV=test

Architecture of the CSA Application

This section will go into detail about the overall architecture of the CSA Application as a whole with a main focus on how the MVC design pattern is represented by describing the key files used within the application itself.

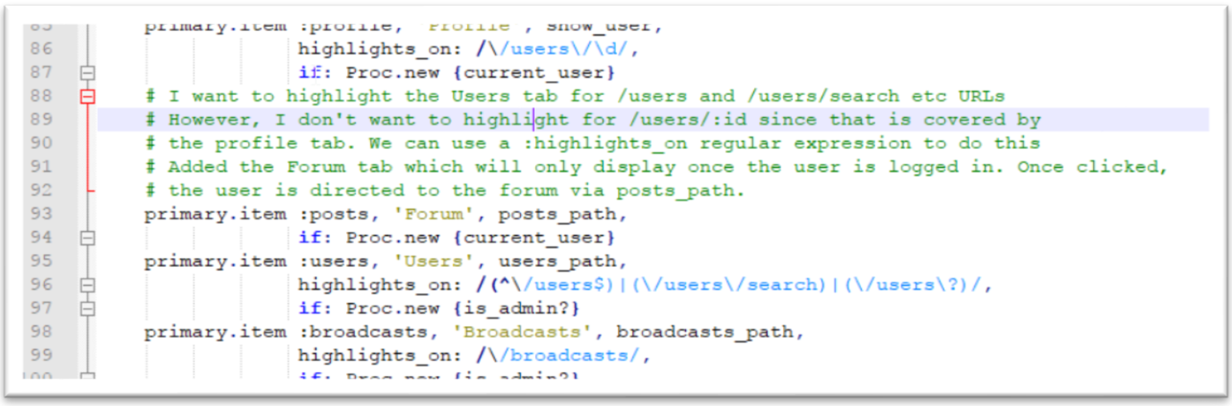
Initially, it would be wise to look at the key files which are not included within the MVC model sections of the application but are nonetheless important in ensuring that the CSA application works as intended. Within the 'config' folder the navigation.rb file allows the user to navigate through the application by using the different tabs that are displayed visually to the user whilst the routes.rb file not only ensures that the user starts on the home page, it also allows the user to search for specific fields through the REST API client.

Meanwhile, the 'db' folder contains the schema.rb which represents the overall structure of the database (which includes the users, user details, forum threads etc). This folder also contains the necessary migrations files that are needed to create and manage any data that is created within the application. It's important to mention that both remove_login_from_posts.rb and remove_column_from_posts.rb simply removes two columns from the posts database that in the end were unnecessary to completing the assignment.

The vast majority of the CSA application architecture is found within the 'app' folder of the CSA folder. Like in the previous Weblog assignment, the 'assets' folder primarily focuses on storing the images and stylesheets needed in order to make the CSA applications look more presentable whereas the 'controllers' folder contains all of the controller classes needed in order to handle the HTTP requests made by the user whenever they wish to interact with the database in some way. The difference that separates the CSA application from the Weblog application however is that the CSA application also contains an 'api' folder that contains different versions of the controllers found within the 'controllers' folder. For the purpose of this section, we will be focusing on both of the posts_controller.rb files and any relevant files that also relate to the Forum feature later on within this section. The main difference between the two files is that the API version of posts_controller.rb only formats to .json whereas the non-API version formats to .html in order to separate the programmable web from the human web. The 'models' folder of the application contains the .rb files needed to give their respective tables the properties that they need in order to function correctly. For post.rb in particular, there is a self referential relationship with the parent_post value which allows posts to be linked to one another in a parent-child fashion. Finally, the 'views' folder contains both the .html files and .json files within the api subfolder needed in order manage and display the information about posts to the user whether it is through the web or through the REST API client. Likewise, the index and show files for both API and non-API versions of the format to .json and .html respectively.

Architecture of the Forum feature

The first part of the assignment tasks us with the creation of the forum feature as a new tab that users can access within the application. It states that the tab must only be exposed to users if they are logged in which means that I had to add the forum the navigation.rb files to allow the creation of the tab and to ensure that the tab would send the user to the correct place when clicked.



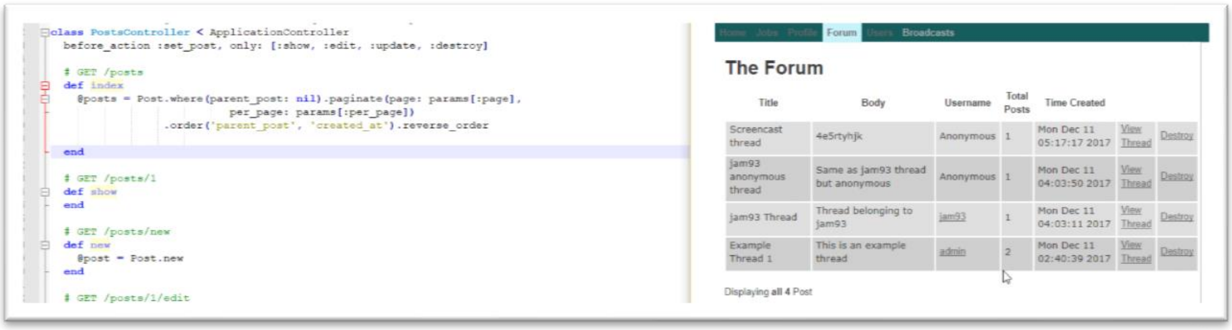
```
    primary.item :profile, 'Profile', show_user,
                 highlights_on: /\/users\/\d/,
                 if: Proc.new {current_user}
    # I want to highlight the Users tab for /users and /users/search etc URLs
    # However, I don't want to highlight for /users/:id since that is covered by
    # the profile tab. We can use a :highlights_on regular expression to do this
    # Added the Forum tab which will only display once the user is logged in. Once clicked,
    # the user is directed to the forum via posts_path.
    primary.item :posts, 'Forum', posts_path,
                 if: Proc.new {current_user}
    primary.item :users, 'Users', users_path,
                 highlights_on: /(^\/users$)|(\/users\/search)|(\/users\?)/,
                 if: Proc.new {is_admin?}
    primary.item :broadcasts, 'Broadcasts', broadcasts_path,
                 highlights_on: /\/broadcasts/,
                 if: Proc.new {is_admin?}
```

Figure 1. Code showing the creation of the forum tab.

Once the tab was in place, I then had to ensure that the following features were met. Listed below is each feature along with my comments on how I implemented said feature:

1. The forum front page must display a list of threads in date-time order with the most recent at the top

This was implemented by editing the posts_controller.rb file to state that the index method uses an SQL statement which sends a query to find every main thread within the posts table and order them by created_at (an automatically generated field which displays the time and date of when the post was created). The order is then reversed so that the most recent threads were at the top and the oldest were at the bottom.



```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # GET /posts
  def index
    @posts = Post.where(parent_post: nil).paginate(page: params[:page],
                                                   per_page: params[:per_page])
                                        .order('parent_post', 'created_at').reverse_order

  end

  # GET /posts/1
  def show
  end

  # GET /posts/new
  def new
    @post = Post.new
  end

  # GET /posts/1/edit
```

The Forum

Title	Body	Username	Total Posts	Time Created		
Screencast thread	4e5rtyhjk	Anonymous	1	Mon Dec 11 05:17:17 2017	View Thread	Destroy
jam93 anonymous thread	Same as jam93 thread but anonymous	Anonymous	1	Mon Dec 11 04:03:50 2017	View Thread	Destroy
jam93 Thread	Thread belonging to jam93	jam93	1	Mon Dec 11 04:03:11 2017	View Thread	Destroy
Example Thread 1	This is an example thread	admin	2	Mon Dec 11 02:40:39 2017	View Thread	Destroy

Displaying all 4 Post

Figure 2. Code showing the SQL statement and the results of the respective code.

2. The forum front page should have a “Create Thread” button.

Within the index.html.erb file of the posts folder, I added a piece of ruby code that create a link that allows the user to create a new thread. This is achieved by making sure that the above link sends the user to the create thread form via using the new_post_path variable.

3. Each entry should display the date and time, title, body, author, unread posts and total posts.

Within the index.html.erb file, the table I created calls the necessary fields from the posts table in order to visually show the above data to the user. A poster variable is used in order to join the user details table to the posts table so that the user’s login info can be displayed as the author of the post. A recursive method is used in order to calculate the total number of posts there are in a given thread, which is then used to display the integer value of the total posts. I omitted the unread posts part of the entry as I didn’t know how to record this information within Ruby and I felt that it would be too time consuming for me to learn how to do this. Finally, a Boolean value is used to determine if the user wishes to make their username public or if they would rather make their post anonymously. An if statement is used in order to check the Boolean value and depending on the result, the entry’s username is displayed or the text “Anonymous” is displayed instead.

4. The list of threads must support pagination.

A div class is made towards the bottom of the index.html.erb file to allow for pagination to happen on the front page of the forum. The def index method within the posts_controller.rb file has also been updated to allow for pagination to occur.

5. An individual thread can only be deleted by the user who created it or by the admin

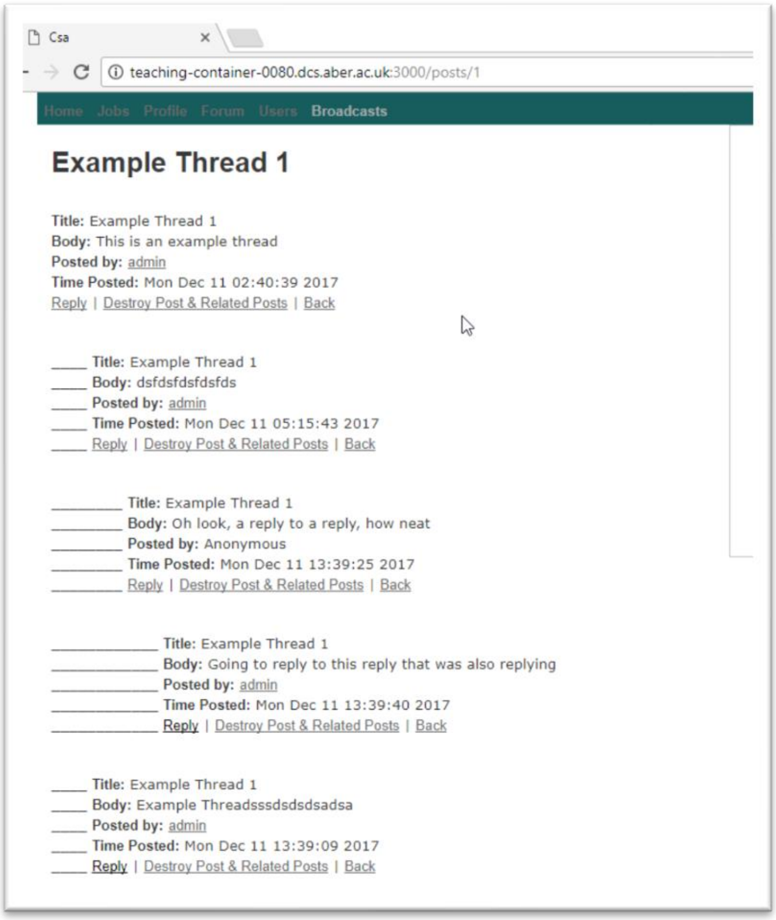
Within the index.html.erb file, a link is created to allow the file to be deleted from the table permanently. This link however only shows if the posts’ username is equal to that of the current user logged in or if the current user logged in is the admin user. Due to the self-referential parent-child relationship within the posts table, a cascade delete SQL statement is applied to all other posts that share the same parent_ID as the ID of the post being deleted. This part is shown extensively within the screencast.

6. When “Create Thread” is clicked, the user is taken to a different page. Title and Body can be filled in, and allow for anonymity.

Within the new.html.erb file, the _form.html.erb file is called to allow the user to create their thread. This form consists of two text inputs to allow the user to type in their desired title and body fields and a checkbox to allow the user to choose if they wish to post anonymously or not. At the end of the form there is a button that allows the user to submit the form as well as a link to allow the user to return to the previous page that they were on.

7. Viewing the thread takes the user to the thread details page.

When the user clicks on the “View Thread” link that is generated within the index.html.erb file for the posts view. The user is taken to a list that consists of the main thread along with all of the replies that are related to that thread. A recursive method is used within the show.html.erb file in order to efficiently ouput each reply visually towards the user.


Csa
teaching-container-0080.dcs.aber.ac.uk:3000/posts/1
Home Jobs Profile Forum Users Broadcasts

Example Thread 1

```
Title: Example Thread 1
Body: This is an example thread
Posted by: admin
Time Posted: Mon Dec 11 02:40:39 2017
Reply | Destroy Post & Related Posts | Back


_____ Title: Example Thread 1
_____ Body: dsfdsfdsfdsfds
_____ Posted by: admin
_____ Time Posted: Mon Dec 11 05:15:43 2017
_____ Reply | Destroy Post & Related Posts | Back


__________ Title: Example Thread 1
__________ Body: Oh look, a reply to a reply, how neat
__________ Posted by: Anonymous
__________ Time Posted: Mon Dec 11 13:39:25 2017
__________ Reply | Destroy Post & Related Posts | Back


_______________ Title: Example Thread 1
_______________ Body: Going to reply to this reply that was also replying
_______________ Posted by: admin
_______________ Time Posted: Mon Dec 11 13:39:40 2017
_______________ Reply | Destroy Post & Related Posts | Back


_____ Title: Example Thread 1
_____ Body: Example Threadsssdsdsdsadsa
_____ Posted by: admin
_____ Time Posted: Mon Dec 11 13:39:09 2017
_____ Reply | Destroy Post & Related Posts | Back
```

Figure 3. Output of a thread with 4 replies.

UML Diagram (MVC for HTTP)

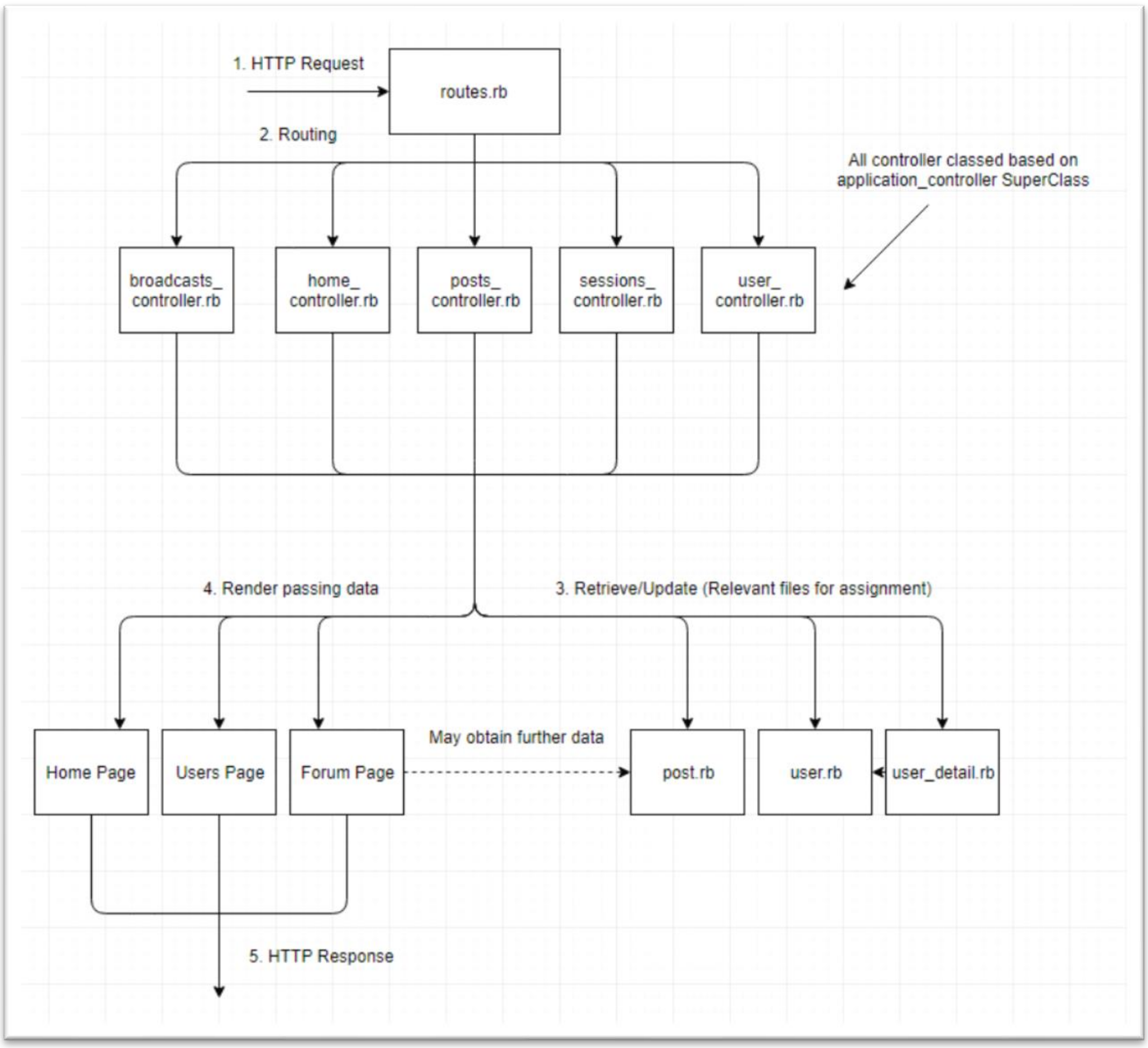


Figure 4. The UML diagram showcasing the MVC for the web application.

UML Diagram (MVC for REST API Client)

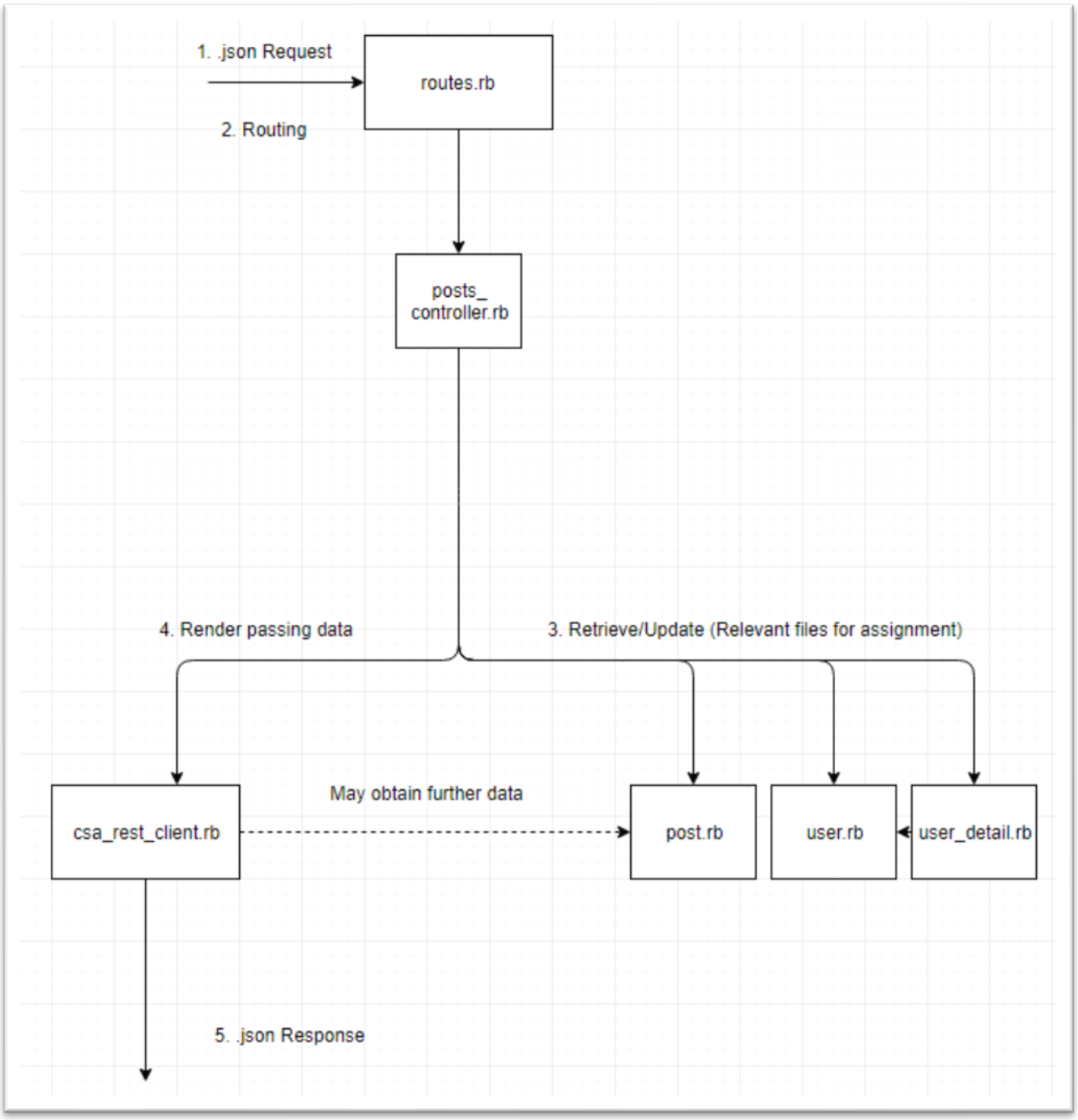


Figure 5. The UML diagram showcasing the MVC for the REST API Client.

Architecture of RESTful client

The third part of the assignment asks us to develop a command-line web service client that issues RESTful requests to the CSA application. The 'rest_client' folder consists of a single file that is named csa_rest_client.rb which handles the creation of the web service that allows the user to send RESTful requests to the CSA application. Initially, this file handled the user's request in conjunction with the Users table within the database thus meaning that the client file was initially designed to allow the user to manage, create and edit the information relating to the users of the CSA application. Since then, the csa_rest_client.rb file has been altered by me in order to allow the web service API to support the forum feature that I have discussed previously within this report.

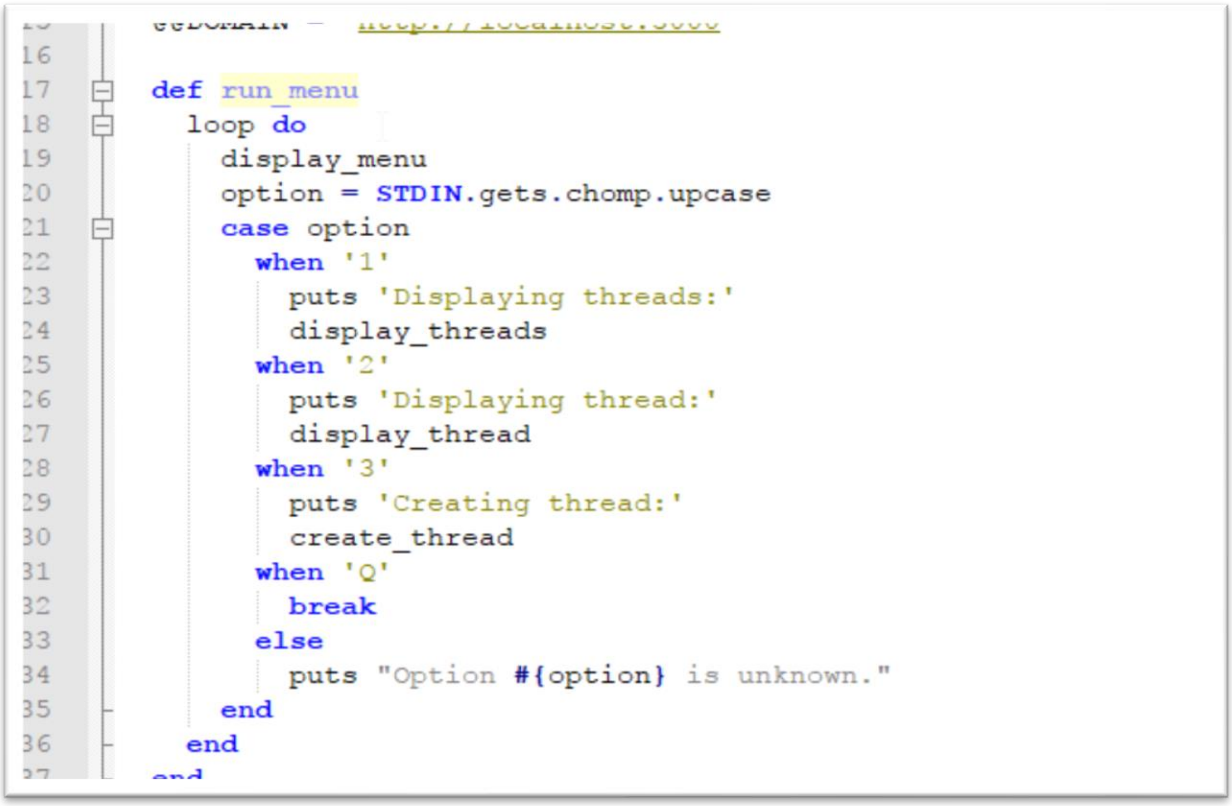
Within this Ruby file, there are 6 total methods that are used at different points of the client which can be seen below:

1. run_menu
2. display_menu
3. display_threads
4. display_thread
5. create_thread
6. authorisation_hash

Initially a CSARestClient object is created which then runs the first method in order to effectively start the client in the user's eyes.

Method run_menu

The run_menu method consist of the different actions that occur depending on which key the user inputs into the client. In a sense, the run_menu method controls the client as a whole. Before the user can input a key, it calls display_menu in order to visually show the command line interface.

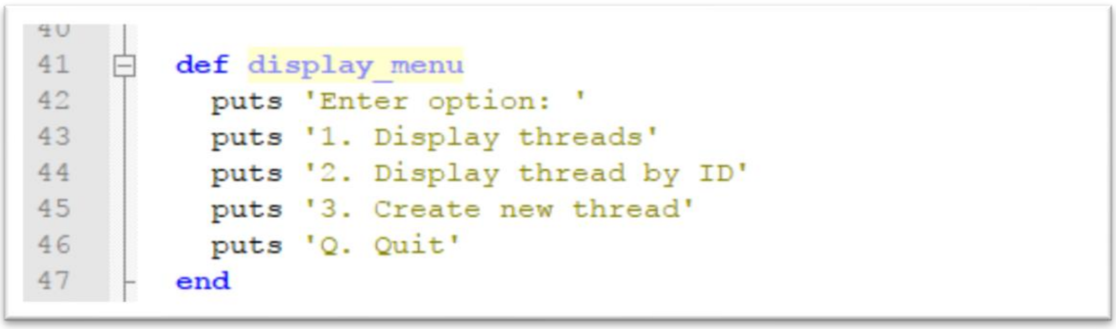


```ruby

def run_menu
  loop do
    display_menu
    option = STDIN.gets.chomp.upcase
    case option
      when '1'
        puts 'Displaying threads:'
        display_threads
      when '2'
        puts 'Displaying thread:'
        display_thread
      when '3'
        puts 'Creating thread:'
        create_thread
      when 'Q'
        break
      else
        puts "Option #{option} is unknown."
    end
  end
end
```

Figure 6. The run_menu method used within the csa_rest_client.rb file.

Method display_menu

This method simply displays the menu to the user by outputting the text onto the command line.

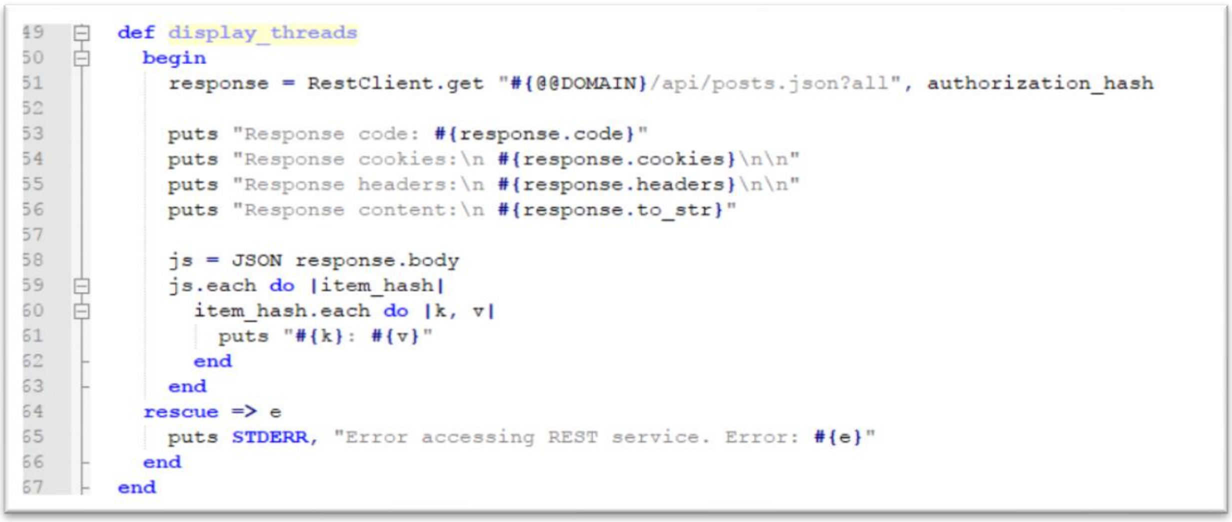


```ruby
def display_menu
  puts 'Enter option: '
  puts '1. Display threads'
  puts '2. Display thread by ID'
  puts '3. Create new thread'
  puts 'Q. Quit'
end
```

Figure 7. The display_menu method used within the csa_rest_client.rb file.

Method display_threads

This method makes use of the modes/api and controllers/api sections of the CSA application in order to output the main threads within the REST client.

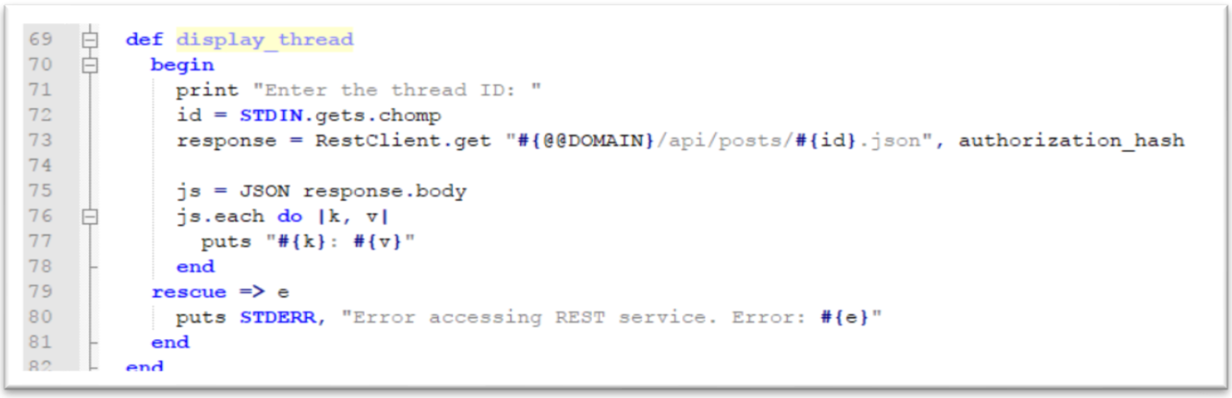


```ruby
def display_threads
  begin
    response = RestClient.get "#{@@DOMAIN}/api/posts.json?all", authorization_hash

    puts "Response code: #{response.code}"
    puts "Response cookies:\n #{response.cookies}\n\n"
    puts "Response headers:\n #{response.headers}\n\n"
    puts "Response content:\n #{response.to_str}"

    js = JSON response.body
    js.each do |item_hash|
      item_hash.each do |k, v|
        puts "#{k}: #{v}"
      end
    end
  rescue => e
    puts STDERR, "Error accessing REST service. Error: #{e}"
  end
end
```

Figure 8. The display_threads method used within the csa_rest_client.rb file.

Method display_thread

The display_thread method allows the user to input an integer value in order to send an SQL query to search the posts table to find a post with the same ID as the integer value typed in by the user. If there is a match then the method will output all of the relevant information to the user.



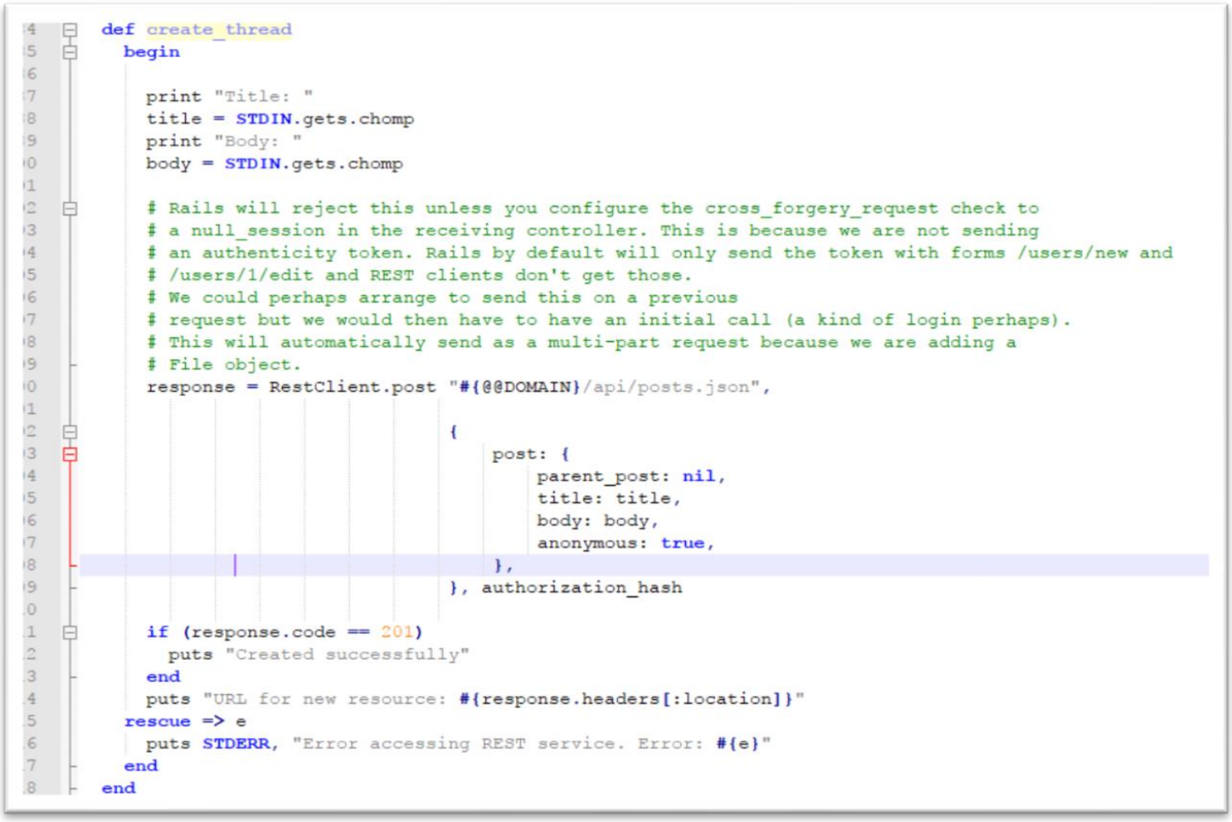
```ruby
def display_thread
  begin
    print "Enter the thread ID: "
    id = STDIN.gets.chomp
    response = RestClient.get "#{@@DOMAIN}/api/posts/#{id}.json", authorization_hash

    js = JSON response.body
    js.each do |k, v|
      puts "#{k}: #{v}"
    end
  rescue => e
    puts STDERR, "Error accessing REST service. Error: #{e}"
  end
end
```

Figure 9. The display_thread method used within the csa_rest_client.rb file.

Method create_thread(Unfinished)

The following method allows the user to create a thread through the REST client by following the instructions displayed on the command line interface. By doing this, the file stores the data that the user types in as variables which are then used in order to create the thread with the stored data.

However, the final version of this method is still incomplete due to the fact that the CSRF token can not be verified whenever the entry is attempted to be stored within the posts table. Below is the code relating to the creation of the thread to show the work that I have created on the method.

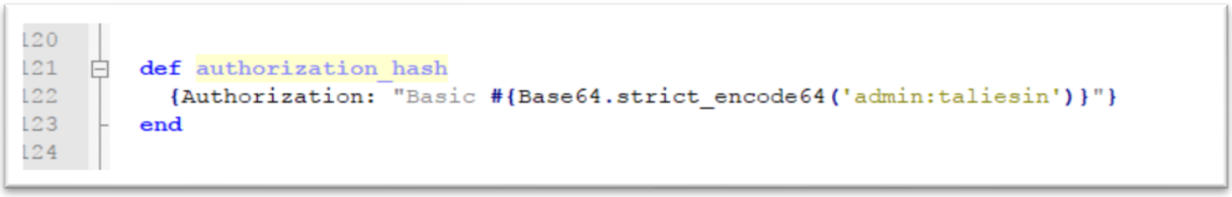


```ruby
def create_thread
  begin

    print "Title: "
    title = STDIN.gets.chomp
    print "Body: "
    body = STDIN.gets.chomp

    # Rails will reject this unless you configure the cross_forgery_request check to
    # a null_session in the receiving controller. This is because we are not sending
    # an authenticity token. Rails by default will only send the token with forms /users/new and
    # /users/1/edit and REST clients don't get those.
    # We could perhaps arrange to send this on a previous
    # request but we would then have to have an initial call (a kind of login perhaps).
    # This will automatically send as a multi-part request because we are adding a
    # File object.
    response = RestClient.post "#{@@DOMAIN}/api/posts.json",

                            {
                              post: {
                                parent_post: nil,
                                title: title,
                                body: body,
                                anonymous: true,
                              },
                            }, authorization_hash

    if (response.code == 201)
      puts "Created successfully"
    end
    puts "URL for new resource: #{response.headers[:location]}"
  rescue => e
    puts STDERR, "Error accessing REST service. Error: #{e}"
  end
end
```

Figure 10. The create_thread method used within the csa_rest_client.rb file

Method authorisation_hash

This method defines the authorisation hash used within the csa_rest_client.rb file.

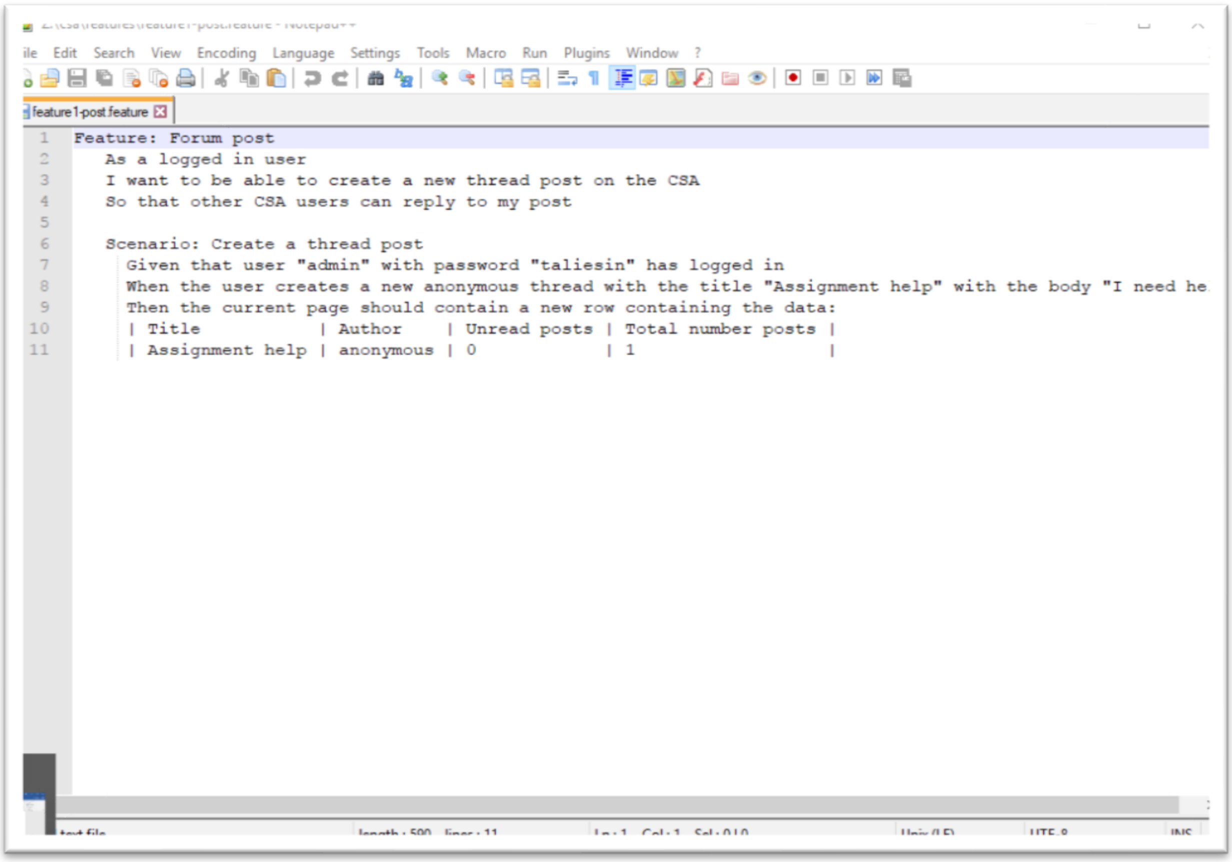


```ruby
def authorization_hash
  {Authorization: "Basic #{Base64.strict_encode64('admin:taliesin')}"}
end
```

Figure 11. The authorisation_has method used within the csa_rest_client.rb file.

Testing with the provided Cucumber feature file

The second part of the assignment asks us to provide stepped definitions in order to test different aspects of the forum functionality. Given to us was the following feature file that can be seen below:


ile Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
feature1-post.feature

```
Feature: Forum post
   As a logged in user
   I want to be able to create a new thread post on the CSA
   So that other CSA users can reply to my post

   Scenario: Create a thread post
     Given that user "admin" with password "taliesin" has logged in
     When the user creates a new anonymous thread with the title "Assignment help" with the body "I need he
     Then the current page should contain a new row containing the data:
     | Title           | Author     | Unread posts | Total number posts |
     | Assignment help | anonymous  | 0            | 1                  |
```

Figure 12. The Cucumber feature file given.

From the above file, we can break the feature down into several steps:

Given:

- That I access the login page
- I type admin into the login text field
- I type Taliesin into the password text field
- I click the sign in button

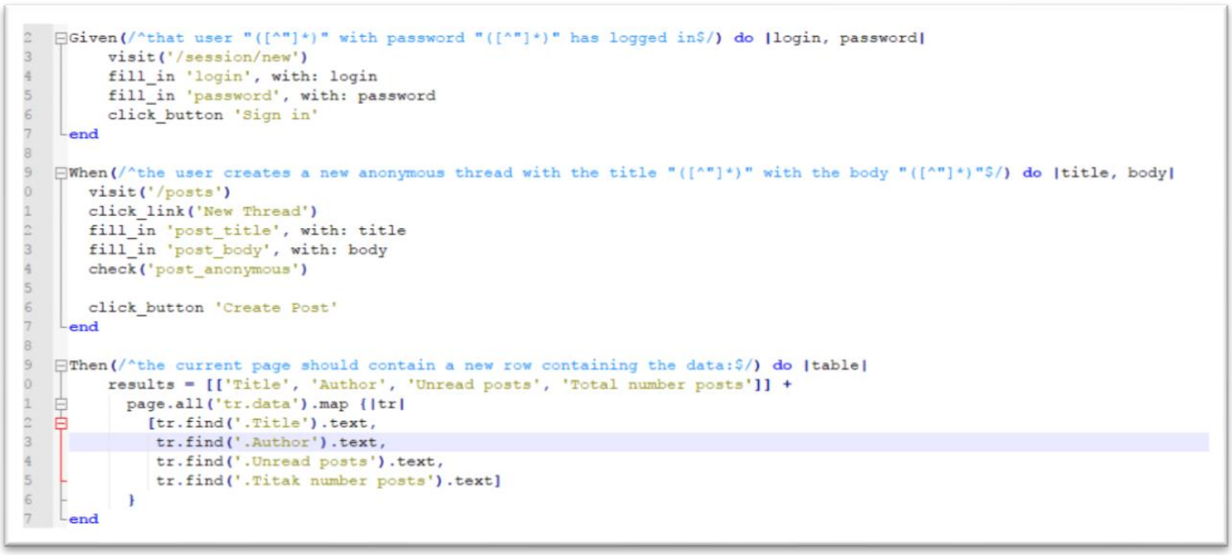
When:

- I visit the Forum tab
- I click on the New Thread link
- I type Assignment help into the title text field
- I type I need help into the body text_area field
- I check the anonymous check box
- I click the Create Post button

Then

- The table will look like the results above

Below is the code that I have written for the steps file:



```ruby
Given(/^that user "([^"]*)" with password "([^"]*)" has logged in$/) do |login, password|
    visit('/session/new')
    fill_in 'login', with: login
    fill_in 'password', with: password
    click_button 'Sign in'
end

When(/^the user creates a new anonymous thread with the title "([^"]*)" with the body "([^"]*)"$/) do |title, body|
  visit('/posts')
  click_link('New Thread')
  fill_in 'post_title', with: title
  fill_in 'post_body', with: body
  check('post_anonymous')

  click_button 'Create Post'
end

Then(/^the current page should contain a new row containing the data:$/) do |table|
    results = [['Title', 'Author', 'Unread posts', 'Total number posts']] +
      page.all('tr.data').map {|tr|
        [tr.find('.Title').text,
         tr.find('.Author').text,
         tr.find('.Unread posts').text,
         tr.find('.Titak number posts').text]
      }
end
```

Figure 13. Code for user_steps.rb

Evaluation

Overall, I feel that the assignment went fairly well for the vast majority of it. Out of the three parts, I found the creation of the forum feature to be the easiest task to do due to the fact that I've had experience using Ruby on Rails from the past Workshops as well as being very used to the overall structure of the MVC from part 1 of this assignment. I particularly enjoyed trying to figure out the best way to link the threads and the replies together and was happy to find out that it was possible via using a recursive method in order to give each post a parent-child relationship using a self-referential model. It took me a fair while to figure out how to properly give indentations to each individual reply as I was initially constructing the output of the posts within the post.children.each do |child| loop using two string variables to handle the indentation.

Whilst the Cucumber testing section of the assignment was difficult at first, it became easier as I had gotten used to the overall syntax used within Cucumber/capybara. This was likely due to only having mild experience using Cucumber in one Workshop practical for this module as well as Cucumber testing not being present in part 1 of this assignment.

The development of the simple REST API web service was personally the most challenging aspect of this assignment for me personally. Although the restructuring of the CSA application was fairly straightforward, trying to understand the error messages that are sent was surprisingly difficult. I managed to get the view thread/s methods working but in the end I did not manage to finish the create_thread method.

The only two omissions I made during this assignment were the create_thread method as mentioned above and having the forum be able to display to the user the number of unread post there are in a given thread.

I feel that I should be awarded 60-65/100 (60-65%) marks for the work that I have created for this assignment. I personally believe that I have completed all of the above tasks to a good quality and I feel that the methodologies that I have used in order to solve these problems were appropriate. However, omitting one of the fields for the first aspect and not being able to fully finish the third aspect will result in me losing a few marks.