Creating a Full App

1. Start with rails new, fill in the gemfile, bundle install, initialize git repository, add readme, setup GitHub/Bitbucket and Heroku, push.

2. Generate a static pages controller, add actions for each page, add routes in the routes file with named routes if necessary. Setup the initial tests. Create preliminary dynamic titles using erb.

3. Set up the title helper method in the application helper.

4. Start with a wireframe for your views, use link_to and name views (which you can see using bundle exec rake routes), use the more semantic header, nav, section, and footer for HTML5 divisions, add images in app/assets/images and use the image tag to find them, add custom site wid styleing and add the include rules for the css files in application.css.

5. Add view partials with __ before the file name and include in the views/layouts folder. Use render to show them.
6. Use Sass (or scss) to simplify CSS with nesting, variables and mixins. Add layout links and test.

7. Follow REST architecture guidelines in controllers and routes, generate a controller for Users with a new page for sign up with a named route.

8. create a user model that includes id, name, email (email index), created*at, updated*at, and password_digest (the timestamps do not need to be explicitly called in the model generation but it is a method that can be used within a migration).

9. Add the following validations:
   - name: presence and length
   - email: presence, length, format, and uniqueness
   - password: has secure password
10. Further ensure email uniqueness by downcasing the email before saving
11. Add a unique index to the email column (use add_index method in the migration)
12. Add the bcrypt gem to the gemfile

13. Add tests for each validation using assert (assert_not) and valid?

14. Users and models in general can be represented in the routes file using resource which automatically creates the REST routes corresponding to CRUD actions on a user

15. Each action needs a corresponding controller action that generally creates an @user instance variable using the params hash and renders a view using this information.
16. Signup page corresponds to the users#new action action in the controller. User form_for(@user) to create a form that when submitted maps the values input to the user model and corresponds to post /users which maps to the action users#create
17. To limit the allowed values for a user submission from the signup page, create a private helper method in the users controller which only allows certain user attributes to be initialized using the create method.
18. For errors, create a helper view in app/views/shared and conditionally render the errors on the base application layout view.
19. On successful signup use the flash object to create a temporary flash display.

20. Pro deployment - use ssl by uncommenting the line in production.rb and use a different webserver (Puma - follow heroku Puma directions)

21. To give the user the ability to login and logout we use sessions. To be RESTful, sessions require a

controller with new, create, and destroy. New maps to the actual sign in form page, create maps to the actual action after clicking login (which redirects to a users show page conditionally if the user is successfully logged in), and destroy maps to logout, which redirects to the root url in this case, allowing a user to log back in.

22. All of these require login and logout links, which can be rendered conditionally.
23. First the form submitted (using the nested hash received by params[:sessions]) must be authenticated. Invalid sign in requires a flash to signify the invalid input valid sign up requires a redirect to the users show page, which includes a log out link.
24. The rails method session creates temporary sessions that expire automatically on browser close.

25. Creating a remember me token requires a number of steps:

    1. Add a remember me checkbox to the login form which submits with the params
    2. Create a persistent cookie if the user selects to be remembered. To create a persistent cookie we:
        1. Add a remember*digest to the user model which is encrypted*
        2. *Store the user id securely (by encrypting it using the signed method) and the remember*token in the cookie using the permanent method.
        3. Check that a user is authenticated by comparing the remember*digest (in the database) with the remember*token (in the cookie)
    3. Forget a user if the box is unchecked by removing the cookie
    4. Conditionally check if a user is logged in on log out

26. To finish the CRUD actions for users we need to be able to edit, which uses a similar form for as sign up.

27. For edit, we need to restrict the edit action to logged in users and to only edit their own account, which requires redirects if these requirements aren't met.
28. Legitimate unsuccessful edits require redirects similar to sign up. The two other types of edits require friendly forwarding which requires a method to store the users current location and redirect back to this on illegitimate edit attempts.
29. We also create a users index which uses pagination to display all of the users, the associated controller action which creates and instance variable users and uses the built in Rails method paginate, seeding the database with fake users using faker, and creating a view partial and the view, which uses will_paginate.

30. The destroy action requires creating admin users by adding a boolean field to the model, creating a conditional link on the index view, and creating a before destroy filter to call an admin_user method to check if a user is logged in as admin.

31. Account activation requires a controller and a resource in the routing file with new, create, edit, and update actions, an activated boolean and an activated at timestamp on the model, and a mailer with both an account activation mailer and a password reset mailer.

32. The mailer controller needs an instance variable for the user information to populate the mailer using the edit action, and the user controller needs the create action to send the mailer, which also requires a change to the redirect for the signup page submission.
33. To activate the account we need an authenticated? method in the user model and then only allow activated users to login.
34. For a password reset, we need to create a controller with just a new and an edit action, but new, create edit and update routes. We also add a reset digest and a reset sent at timestamp on the model. Then we change the login view to have a reset password link and we create the reset password view.
35. Then we create the actual password reset mailer by adding the mailer controller action, filling out the mailer views, and defining the password reset controller actions. The user model also needs methods to create the reset digest, send the password reset email, and check if the password reset

link is expired. The final action is to fill out the password resets view linked to the edit view.

36. The config/environments/production needs to be set up correctly to send emails in production.

37. Microposts require a model with content, a user id, and a timestamp. The user id is created by making a reference to the user class, which generates a foreign key also. The model needs validations for length of content and presence and presence of a user id, and create the association (belongs to) with a user. Listing them in order requires a default scope to be specified to list them in reverse order using the desc command. They should be dependent destroy with users.

38. Showing microposts is done on the user's show page using paginate and requires the controller actions create and destroy. This requires the form and the error messages partial. We also need delete links for a logged in user to delete posted microposts.

39. To add images to microposts, generate an uploader using CarrierWave, create an image association in the micropost model, and add the upload button to the view. Create size and type validations in the model but also use client-side by testing for type with jQuery and adding an accept parameter in the image tag. Use the fog gem for production image uploading.

40. To create a feed users need to be able to follow and unfollow other users. To model this, rails g a new model with an id, follower id, followed id, and timestamps, add unique indices to both the follower id, the followed id, and the relationship between follower and followed to avoid duplicated following.

41. Separate the relationships between active and passive and create belongs to associations in the user model. Use the source parameter to specify separate classes so the associations can be renamed for ease of understanding. Create validations in the relationship model for presence of the follower/followeds ids, and create a has many through association to create the following relationship (i.e. a user has many following users through the relationship model). Then create a similar set of specs for the passive_relationship, i.e. a user being followed by another user.
42. Create a stats view to show the number of users following and followed, create buttons/forms for following and unfollowing, create the paginated views to show the following users, and complete the simple html form action.
43. Add remote: true to allow ajax for the follow form submission, enable ajax in the application configuration, and create the js.erb files with in the folder with the view with the same name as the action to write the ajax actions.
44. To finish implementing the feed, reimplement it using SQL in rails and by creating a subselect to avoid loading large amounts of data at once.