Homework 3: PennInTouch Lite

Regular credit is due **11:59PM, Thursday, April 16th**.
Extra credit is due on the last day of class.

I. Getting Started

In this homework, you will implement a simple Penn InTouch clone where you can create users (instructors and students) and courses, log in as a user, and add students to courses.

Before you get started, we suggest you take some time to read through this README to understand the features, the models and their associations, boot up the starter code and understand what the starter code has done for you.

After you have cloned this project to your `CIS196` folder, be sure to do the following steps:

1. run `bundle install`. We've modified the `Gemfile` to include a number of third-party gems that you'll work with in this homework including `bootstrap`, `bcrypt` (for user password encryption), `pry` (for debugging).
2. run `rails db:create`, `rails db:migrate` to establish the database. The starter code automatically uses `postgres`, so you don't need to configure it.
3. run `rails db:seed` to populate the `Department` records into the database.
4. run `rails s -b 0.0.0.0` and see what the starter code does for you.

You should see a homepage on `localhost:3000`, where you'll see a navbar. If you click on any of the links, you will get a `No route error`. You'll need to fix them as the homework progresses.

Understanding the Starter Code

It is important to understand what we have done for you:

- We configured bootstrap and some custom CSS in `/assets/stylesheets/application.scss`. These styles follow the Central Penn Web Identity
- We set the root to `pages#home`, created a `PagesController`, and wrote the homepage with HTML `/views/pages/home.html.erb`.
- We provided `/views/layouts/_navbar.html.erb`, a partial used in `/views/layouts/application.html.erb`. You will need to modify this navbar after you have implemented sessions.
- We created the `Department` model. If you had run `rails db:seed`, your database would now contains all 224 departments of Penn. The `Department` has a one-to-many association with `Course`s. It doesn't have controller and views and YOU DONT NEED TO WORRY ABOUT THIS throughout them homework. Just keep in mind that when you create the `Course` model, you need to maintain the one-to-many association.

- We created a `RegistrationController` class in `registration_controller.rb` which contains stubbed code that you need to complete when implementing the many-to-many association between courses and students through registrations. There are no views or CRUD controller methods for `Registration`, so you don't need to worry about them.
- We created a `SessionsController` class in `sessions_controller.rb` which contains contains stubbed code that you need to complete when implementing sessions. There is a `/views/sessions/new.html.erb` where will need to implement the login form.

It is a good idea to do a "Find in Folder" (global search) in VSCode for the keyword `TODO`. This marks all the unimplemented code in the provided files we give you.

Understanding the Features

To understand the expected result of this project, go to my demo and see the final result.

You can try signing up once as an instructor, and a second time as a student. The features available to an "instructor" and a "student" differs a little (as you'll also implement in the last section). You may try creating a course as an instructor, adding a course as a student, log out, log in, etc.

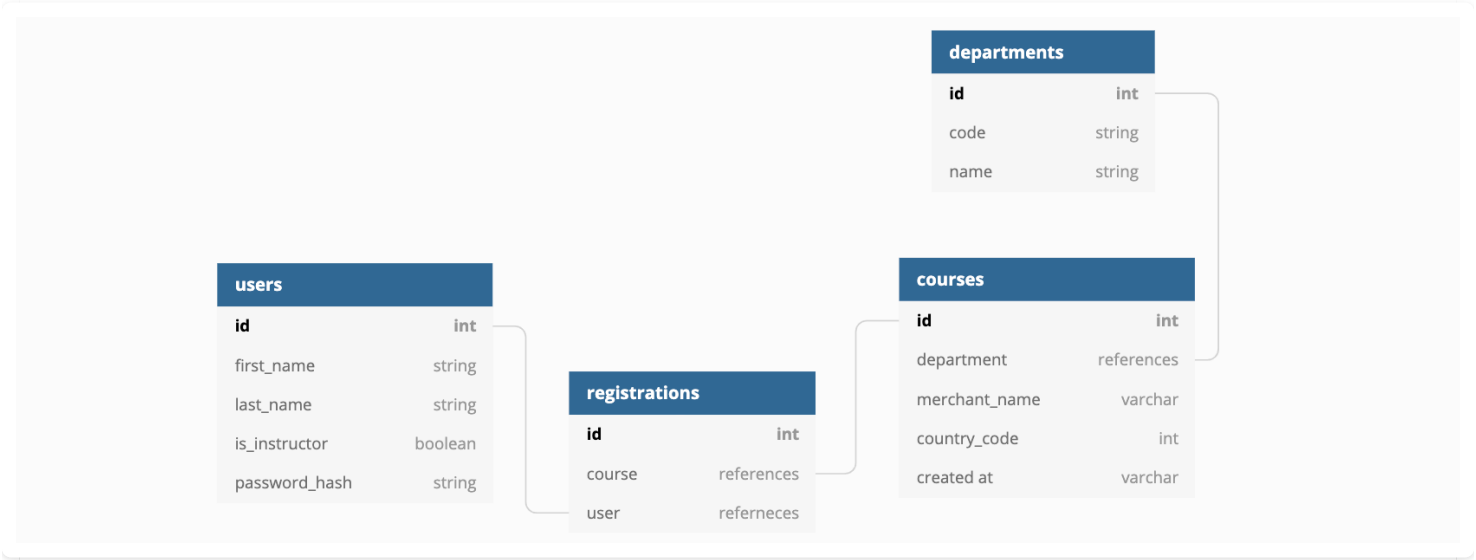
> *Sanity Check*
>
> 1. *run* `rails c` *to check if* `Department.all.count == 224`.
> 2. *run* `rails s -b 0.0.0.0` *and check* `localhost:3000` *that you have a well-styled homepage and navbar.*
> 3. *read through the rest of the doc to understand the models and features you need to implement for this project.*

A word on `rails db:seed`

In this project, you may find yourself in need of dropping your database once in a while. After you rerun `rails db:create` and `rails db:migrate`, make sure you also run `rails db:seed` so that you get all the `Departments` populated into your database from the start.

II. Generating Scaffolds and Models

The first task of this project is for you to figure out the CRUD models. There are 4 models involved: `Department`, `Course`, `User`, `Registration` (which is just a join-table for courses and users). Recall from the lectures that `session` is not a model, but just an in-memory construct. The relationship/association between the four models are as such:



As you can see:

1. `Department` and `Course` is a one-to-many association. Although the starter code have fully implemented `Department`, you are responsible for setting up `Course` and the association.
2. A `User` can be either a student or an instructor, based on the field `is_instructor`.
3. `Courses` and `User`s have a many-to-many association **through** `Registration`, and `Registration` is merely a model/table that stores references to a course and a user.

You should now get started with generating scaffolds and models for these classes using the correct `rails g` commands. Recall that if you made a mistake during `rails g`, you can use `rails destroy` to reverse your action.

Specification for Models

Course

The `Course` model should have a `department` of type `references`, `code` of type `integer`, a `title` of type `string`, and a `description` of type `text`. It needs a controller and views, so it's better to use `scaffold` generator.

User

The `User` model should have `first_name` and `last_name`, `pennkey` columns of type `string`, an `is_instructor` of type `boolean`, and a `password_hash` field to store passwords hashed by `BCrypt`. It needs a controller and views, so it's better to use `scaffold` generator.

Registration

The `Registration` join table should have `references` to a `course` and a `user`. It does NOT need views and CRUD methods in controller, so it's better to use `model` generator. Note that we have provided you with a `registration_controller.rb` where you will be implementing the `add_course` and

`drop_course` feature.

Be sure to run `rails db:migrate` !

After that, make sure you have added the necessary lines to the model files to establish the required associations! Be sure to debug and test these behaviors using `rails c` . Also, be sure to add the `dependent: :destroy` option where appropriate.

> ***Sanity Check***
> *At this point, you should be able to:*
>
> 1. *run the CRUD methods and association queries (* `@course.users, @course.department, etc...` *) in your* `rails c` *environment.*
> 2. *run* `rails s -b 0.0.0.0` *and see that the navbar items of* `Users` *and* `Courses` *no longer gives you a* `No Route Error` *.*

Validations, Arel and Custom Methods

While our models have our desired attributes, they are far from complete. We want to add some validations as well as describe some custom methods for these models to make our lives easier.

Course

1. You should validate that the `department` , `code` , `title` , and `description` are always present.
2. You should implement an instance method called `full_code` that returns a string which combines the department code and the course code for that course. For example: for a course with code `196` whose department has code `CIS` , the method should return `"CIS-196"` . Hint: string interpolation helps.
3. You should validate that checks that the `full_code` of each course is unique. you can use `uniqueness` with `scope` documented here and here.
4. You should implement an instance method called `instructor` , which returns the first user in the course's `users` array whose `is_instructor` field is `true` . Hint: use Arel method `find_by` .
5. You should implement an instance method called `instructor=` , which takes in a `user` as an argument. The method first checks whether the user's `is_instructor` field is `true` , if so, shovel the user into the course's `users` array. This is like a setter method for instructor of a course.
6. You should implement an instance method called `students` , which returns all users in the course;s `users` array whose `is_instructor` field is `false` . Hint: use Arel method `where` .

User

1. You should validate that the `first_name` , `last_name` , `pennkey` , and `password_hash` are always present.
2. You should validate that the `pennkey` is unique for each user.
3. You should implement an instance method called `full_name` that will return a user's full name (i.e.

The user's first and last name separated by a space).

4. You should implement a class method called `students`, which returns all users whose `is_instructor` field is `false`.
5. You should implement a class method called `instructors`, which returns all users whose `is_instructor` field is `true`.
6. You will come back to add a few more methods to enable BCrypt, following the tutorial.

Registration

1. You should validate that every **student(user) and course *pair*** is unique in our join table. Again, you will need to use `uniqueness` with `scope`. See documentation here and here

> *Sanity Check*
>
> 1. *You should test the correctness of your validations using the CRUD operations in* `rails s`
> 2. *You should test the correctness of your custom methods in* `rails c`
> 3. *Towards the due date for this homework, we will release a set of* `rspec` *tests that covers the correctness of your model layer. You should run it to make sure all tests pass.*

Finishing Touch on Course Form

Before we move on, there's one thing we need to fix. After you boot up `rails s`, you will notice that when you try to create a new course, the "Department" field is a text_field, which is not ideal for department selection. Modify `/views/courses/_form.html.erb` to change the `form.text_field` to `form.collection_select`, which generates an HTML `<select>` component.

You will need to dig a little into the API for `form.collection_select`. Note that the `method` refers to the model's field name (`department_id` in this case), the `collection` is the array of department instances to be selected from, the `value_method` is the field of the department instance that you need to assign to `department_id`, the `text_method` is the text to display in the `<select>` component.

III. Sessions and BCrypt

Before we increase the complexity in our features, it is a good idea to first implement user authentication, so that we customize the features based on whether or not a user is logged in.

This section is extensively discussed in lecture 9 so be sure to review the lecture slides.

Integrating BCrypt

Recall that we **never** want to store passwords in plain text -- it's a security nightmare. To address this, we utilize the BCrypt function to help us hash our passwords.

Since Gemfile in the starter code already includes BCrypt, you should have BCrypt already available to you. Follow the "User Model" section of BCrypt documentation to modify `user.rb` model file again. This step is

consistent with what's described in the slides.

To sum up what's been done in this step: First, `user#password`, is a **getter method** that reads from our database and creates a `BCrypt::Password` object where we can directly compare plain-text passwords to. Second, `user#password=`, is a setter method for the `password_hash` field of the user that sets the value to the hashed version of the plain-text password we feed in. Therefore, with these helper methods, you **should never have to use `password_hash` directly**. You may test this out in `rails c`

```
@user = User.new({first_name: .......})
@user.password = 'mypassword' # The setter
@user.password # The getter
# => Some text gibberish, a "hash"
@user.password == 'mypassword'
# => true
@user.password == 'yourpassword'
# => false
```

If the above step works for you, you should make sure you hash the password in the `create` method and `update` method of the `UsersController`.

For the `create` method, after a user is `new`ed, you should call the `password=` set method and feed it with the plain-text password. (Hint: the plaintext password is the `password_hash` field you received from the form, which is now located in `user_params`).

For the `update` method, Instead of directly calling an `update` with the `user_params`, you should `update` down to two steps: `assign_attributes` and `save`, just like `new` and `save` in `create`. After you assign attribute, and before you call save, you should call the `password=` set method again with the new plain-text password before saving to the database.

You may want to use `binding.pry` to set breakpoints in these methods to inspect whether or not passwords are correctly retrieved, hashed, and stored.

Implementing Routes, Views and SessionsController for Authentication.

This part is consistent with the content in the slides of lecture 9. You should set up 3 routes for `/login` and `/logout`, which are handled by the `SessionsController`'s `#new`, `#create`, and `#destroy` methods respectively.

The `new` method handles the `GET` request to `/login`, and uses `/views/sessions/new.html.erb` to display a form. You don't need to modify the `new` method, but be sure to implement the login form in `new.html.erb`.

The login form should have a text field for pennkey, and a text field for password. When the form submits, it should be submitted to `POST /login`, which is handled by the `create` method.

The `create` first find the user given the pennkey submitted. If no user is found, it render `:new` with the

`@message = 'User name not found'`. If it is found, then the method checks whether the user's password equals the submitted password. If so, redirect to the user's homepage, else render `:new` with the `@message = 'User name not found'`.

The `destroy` method just calls `reset_session` and redirects to `/login`.

Adding Helper Methods in ApplicationController

Before we can use `sessions` to customize views for logged-in users, we need several helper methods in `application_controller.rb`. Recall that since all controllers extend `ApplicationController`, these methods are available in all the controllers and views.

Consistent with the lecture slides, you should implement 3 methods:

The `logged_in?` method returns `sessions[:user_id]`. This will be used to customize the navbar later.

The `current_user` fetches the current logged-in user instance using `sessions[:user_id]`.

The `authenticate_user` method checks whether there is a logged-in user. If not, it redirects to "/login" to force a user to log in. This will be used as a `before_action` for our controllers.

Customizing Views and Controllers

Using `logged_in?`, you should modify `/views/layouts/_navbar.html.erb` so that when the user is logged in, only "Users", "Courses", "Log Out" is displayed, and when the user is not logged in, only `Sign Up` and `Log In` is displayed.

You should use `before_action :authenticate_user` on `CoursesController`, `UsersController`, and `RegistrationsController`. All their methods except for `UsersController#new` and `UsersController#create` should run `authenticate_user` before everything else.

For a finishing touch, make sure you:

1. In `UsersController#destroy`, add a `reset_session` call before you redirect. Otherwise your `logged_in?` will return true for a non-retrievable user instance.
2. In `CoursesController#create`, set the course's instructor to the `current_user` **after** saving to the database and before redirecting.

It is also recommended that you change the `text_field` for `password_hash` to a `password_field`, and change the label to just "password".

> ***Sanity Check***
>
> 1. *You should be able to use the difference in navbar to tell whether or not a user is logged in.*
> 2. *You should be able to create a user with a plain-text password, authenticate using that plain-text password, update the password with a new plain-text password in `edit`, log out, and reauthenticate using the new password, **BUT NOT the OLD PASSWORD**.*

IV. Course Registration

We will implement the main feature of PennInTouch in this section, which is add and drop course. We will develop appropriate views to enable these action.

Routes for `RegistrationsController`

You'll need to modify `routes.rb` to add 2 routes, one for adding a course for a student (user), and one for dropping a course for a student (user). The HTTP methods and the paths are completely up to you, although our recommended combination is:

- POST method, for path "/add_course/:user_id/:course_id"
- DELETE method, for path "/drop_course/:user_id/:course_id"

The important thing is that the add-course route is handled by `RegistrationController` 's `add_course` method, and the drop-course route is handled by the other. Another key thing to note is that the path of the route must incapsulate the information needed to retrieve the relevant student (user) and the relevant course.

Methods in `RegistrationsController`

`set_course` and `set_student`

Because both `add_course` and `drop_course` need to find the relevant student instance and the course instance, we provided the space for you to implement the before_actions of set_course and set_student.

`add_course`

To add a course to a student(user) is just shoveling the student(user) instance to the course's `students` array.

This is easy, but be sure to look at the hints in the `TODO` to implement some checking. For example, if the user is an instructor, you should not perform this action.

In the end, don't forget to redirect the user to its own show page.

`delete_student`

To drop a course for a student(user) is just deleting the student(user) instance from the course's `students` array.

This is easy, but be sure to look at the hints in the `TODO` to implement some checking. For example, if the user is an instructor, you should not perform this action.

In the end, don't forget to redirect the user to its own show page.

Views for Course Registration

You will need to extensively modify the views of your current project. Start by removing the `scaffold.scss` from `/assets/stylesheets/`. The final result should look like what we have in the demo. Some major changes are described here:

Course `index` page:

Instead of displaying a table row, you should display a bootstrap card for each course, and each card should take up 6 columns in a 12-column bootstrap row.

The title should be the `full_code` of the course, the card subtitle should be the course's `instructor`, The card's text should be the `description` for the course, but limited to 120 characters.

There should be one or two links for the course card, depending on who the `current_user` is:

- There always need to be a link to the show page of the course.
- If the `current_user` is a student who have not added the course:
  - then there need to be a second link with text "Add Course", which on click, makes a request to the route you defined for `add_course`.
- If the `current_user` is an instructor created the course (is the instructor of the course):
  - then there need to be a second link which links to the "Edit" page of the course.

Moreover, if the `current_user` is an instructor, there should be a button above all the cards that links to the `new` page for courses.

User `show` page:

Instead of displaying all the fields of a user, you should have two sections.

In the `User Info` section, you will only display the user's first name, last name, pennkey, and status (if is_instructor, display "instructor", else display "student").

The title of the second section depend on whether the user is an instructor. If so, display "Courses Teaching", otherwise, display "Courses Taking".

The second section should display all the courses the user is taking or teaching, in the same format of the course cards.

Again, there should be one or two links for the course card, depending on who the `current_user` is:

- There always need to be a link to the show page of the course.
- If the `current_user` is the same user displayed at the show page,
  - If the `current_user` is a student, then there needs to be a second link with text "Drop Course", which on click, makes a request to the route you defined for `drop_course`.
  - If the `current_user` is an instructor, then there needs to be a second link which links to the "Edit" page of the course.

> *Sanity Check*

> *At this point, you should be able to:*
>
> 1. *Log in as an instructor, create a course, and see that your course displays with two links on the index page of courses, the second link is to edit.*
> 2. *Going to your own show page, you should see the course displayed in the* `Courses Teaching` *section with two links.*
> 3. *Log in as another instructor, and see on the previous course created only displays one link on the index page of the course.*
> 4. *Log in as a student, you see the previous course displays with two links on the index page of courses. When you click on the second link, you are registered to the course and redirected to your own show page*
> 5. *On the show page, you can see the course you just registered displayed in the* `Courses Taking` *section with two links. When you click on the second link, you drop the course, and can re-add it from the index page of courses.*

V. Authorization and Miscellaneous Views

For all the other views, you should aim to make them look as similar to the demo as possible, or even better. Although we don't grade harshly on them, but these following points are required for the user `index` page:

- The user `index` page should display two tables, one for all the instructors and one for all the students.
- Do not display the password column, and links for edit and destroy.
- Above both tables, there should be a button that links to the `show` page of the `current_user` .

The above modifications to views only hide the certain routes from clicking access. Dangerously, any student can still edit every other student's information, password, and registration. There are many loopholes that one can exploit. Therefore, we need to add some before_actions in controllers to make sure these scenarios do not happen:

1. A non-logged-in user can only access root page, sign up ("users#new", "users#create"), and log in.
2. A logged-in user cannot edit another user's information.
3. A logged-in user can only register and drop courses for herself, and only a student can register and drop courses.
4. A logged-in student can only Read courses, but cannot create, update, destroy courses.
5. A logged-in instructor can only edit courses created by herself, and only an instructor can create courses.
6. If any of the above is violated by a logged-in user, the user should be redirected to either the course index page or the show page of herself, depending on the circumstances.

To implement these invariants, you will need to use `before_action` on various controllers. You can define the methods used by `before_action` either in `ApplicationController` , or as private methods of certain controllers. Be sure to check for whitelist (`except`) and blacklist (`only`) defined by before_actions.

If you are unsure about certain behaviors, please ask on Piazza or reference the demo.

VI. Deployment and Submission

Same as with HW2, you need to deploy this project on Heroku. Follow the steps in HW2 if you forgot how to do it.

When you submit:

1. stage all changes, commit, and push to gitlab
2. zip up the project and upload it on canvas
3. Along with your canvas submission, post a comment that is the link to your heroku-deployed project.

VII. Extra Credits

Using what have learnt in the lectures, you can go on to implement several features to make PennInTouch Lite richer and more usable.

User Avatar with ActiveStorage (+10%)

Generate a migration to add an `avatar` field to each user. Use active storage to handle file upload and storage of avatar images, and display the avatar image in user's `show` page as well as `index` page.

Add comments to courses (+10%)

Generate another MVC for `Comment`, which must be a nested resource under `Course`. Users can create comments for a course.

Filter course by `Department` (+15%)

Extend the behavior of `/courses` to `/courses?department=[SOME_DEPARTMENT]`, so that on visiting `/courses?department=CIS`, only CIS courses are displayed on the index page.
To enable filtering with clicks, you will need to implement a form on the index page which has only one input: a department. On submission of the form, the index page should refresh (redirect) to extend the path, and load only the courses for that department.
You will also need to handle the edge cases gracefully.

Your own idea (+5% to +20%)

You are welcome to submit your own extra credit ideas and we'll evaluate the point worth.