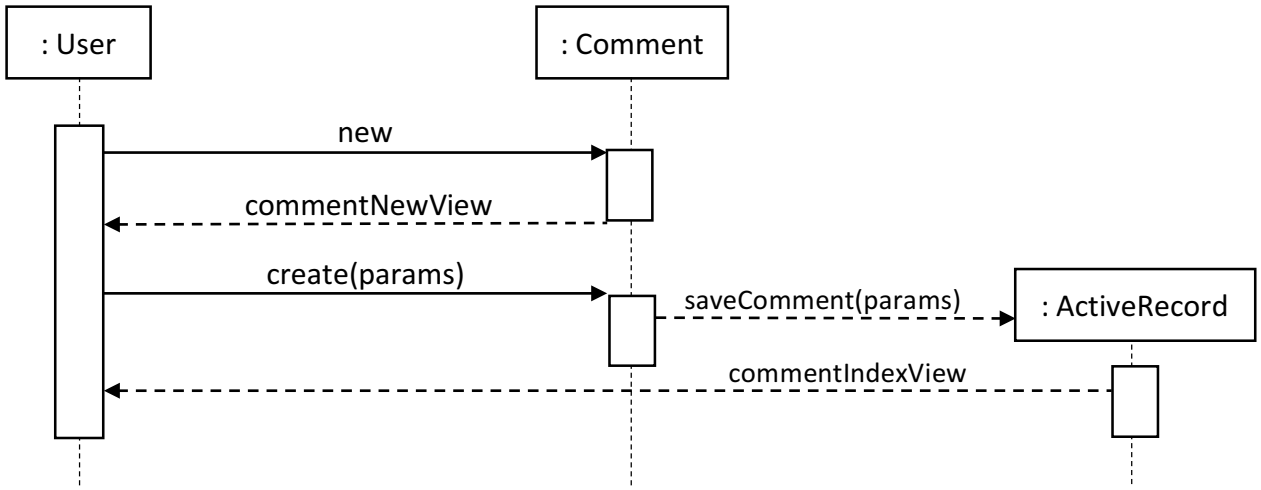
Iteration 3 Design Document

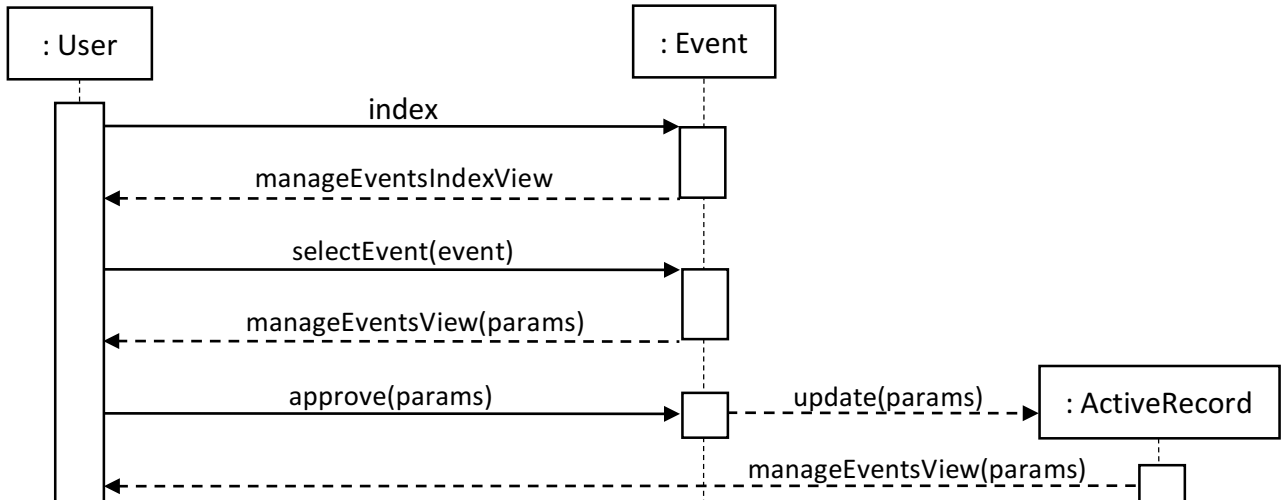
Add Comments (Basic Flow)

1.) User chooses an event from the left sidebar that they want to view the comments for.
2.) User wants to leave a comment for the event, so they enter in their comment in the textbox.
3.) User clicks the Post Comment button.
4.) Comment appears on the page if the event is not moderated.



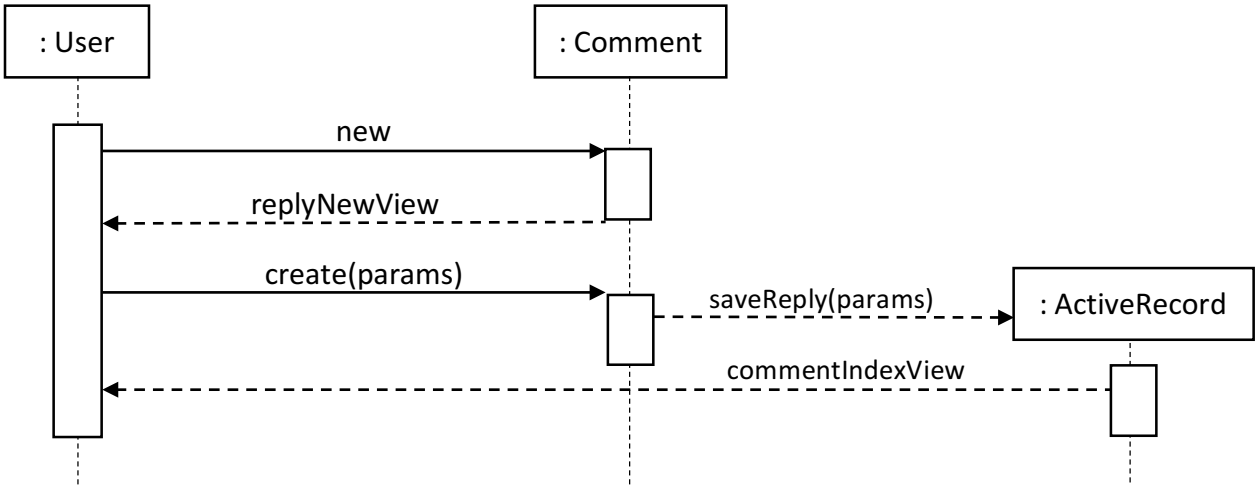
Moderate Events (Basic Flow)

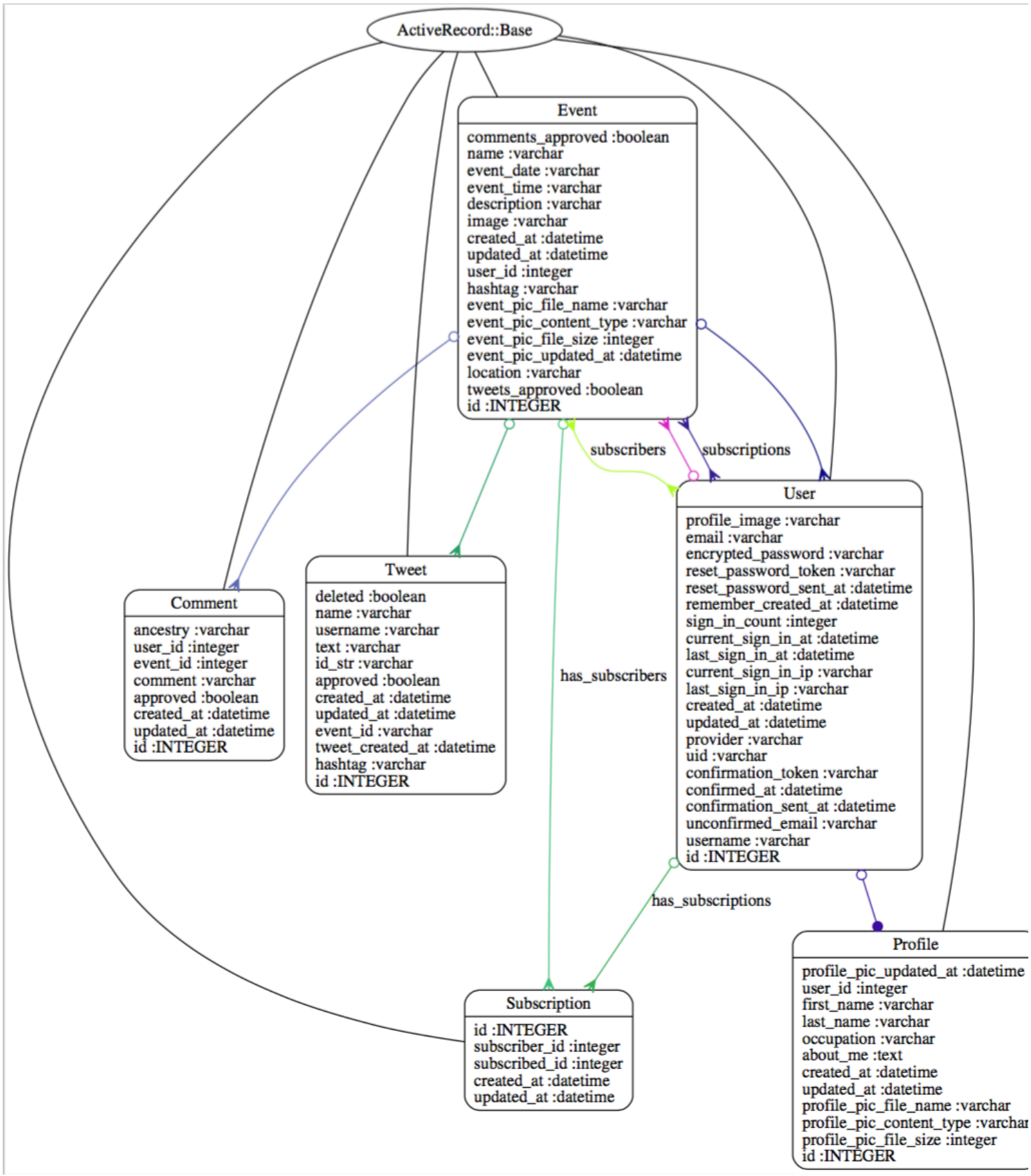
1.) Event Manager clicks on the “Manage Events” button in the dropdown menu in the top navigation bar.
2.) Event manager chooses what event they want to moderate from the sidebar on the left.
3.) All tweets and comments that have not been moderated yet are displayed on the page
4.) Event manager wants to approve a tweet or comment, so they click on the checkmark icon next to the tweet or comment to approve it.

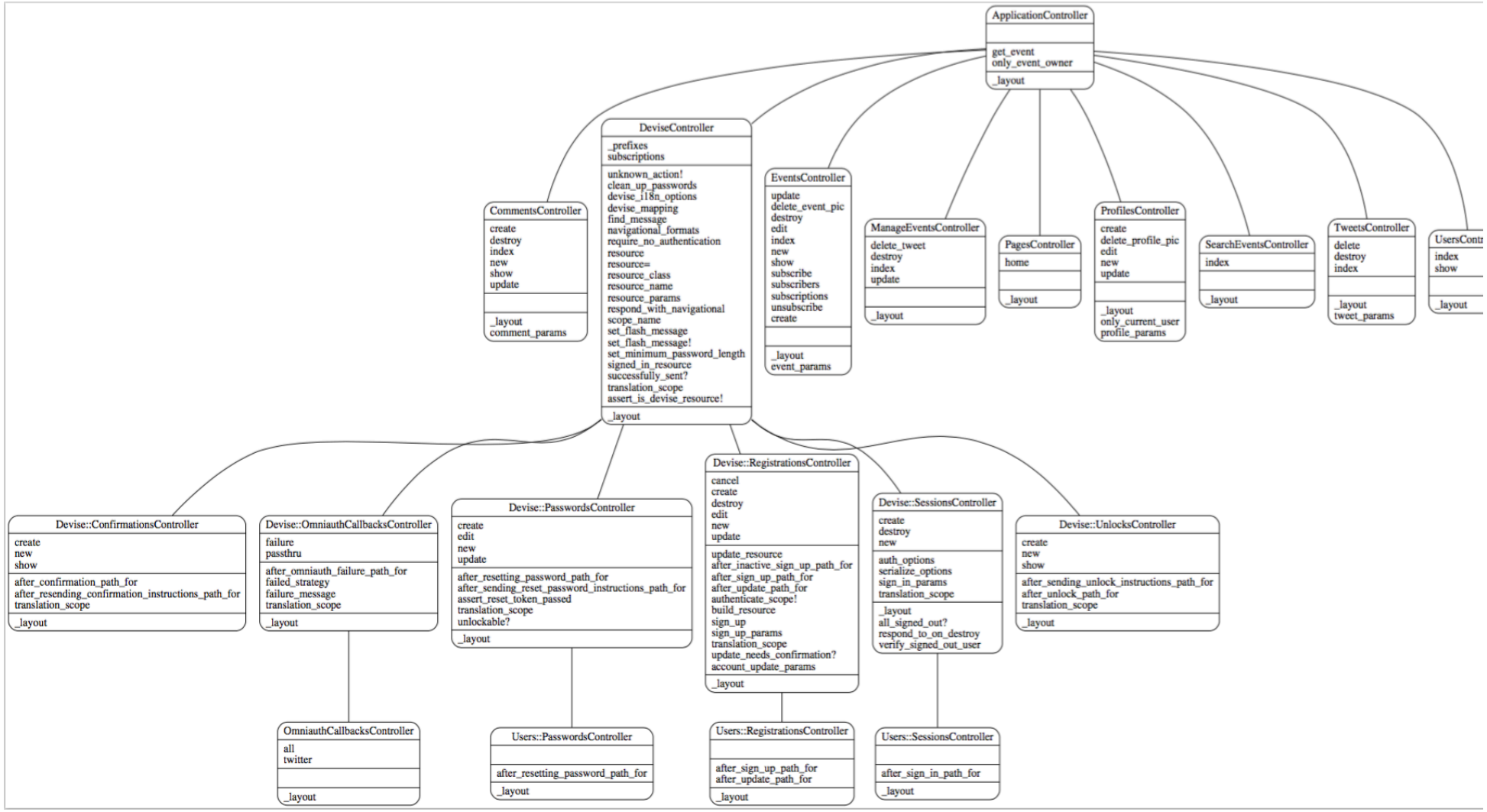


Add Comments Scenario (Alternate Flow)

1.) User navigates to the Create New Event page by clicking one of the Create New Event buttons.
2.) User wants to reply to an existing comment.
3.) User clicks the reply icon on the comment that they want to reply to.
4.) User enters their reply into the textbox.
5.) User clicks the Reply button.
6.) The reply comment appears on the page if the event is not moderated.




ActiveRecord::Base
Event
comments_approved :boolean
name :varchar
event_date :varchar
event_time :varchar
description :varchar
image :varchar
created_at :datetime
updated_at :datetime
user_id :integer
hashtag :varchar
event_pic_file_name :varchar
event_pic_content_type :varchar
event_pic_file_size :integer
event_pic_updated_at :datetime
location :varchar
tweets_approved :boolean
id :INTEGER
subscribers
subscriptions
User
profile_image :varchar
email :varchar
encrypted_password :varchar
reset_password_token :varchar
reset_password_sent_at :datetime
remember_created_at :datetime
sign_in_count :integer
current_sign_in_at :datetime
last_sign_in_at :datetime
current_sign_in_ip :varchar
last_sign_in_ip :varchar
created_at :datetime
updated_at :datetime
provider :varchar
uid :varchar
confirmation_token :varchar
confirmed_at :datetime
confirmation_sent_at :datetime
unconfirmed_email :varchar
username :varchar
id :INTEGER
Tweet
deleted :boolean
name :varchar
username :varchar
text :varchar
id_str :varchar
approved :boolean
created_at :datetime
updated_at :datetime
event_id :varchar
tweet_created_at :datetime
hashtag :varchar
id :INTEGER
Comment
ancestry :varchar
user_id :integer
event_id :integer
comment :varchar
approved :boolean
created_at :datetime
updated_at :datetime
id :INTEGER
has_subscribers
has_subscriptions
Profile
profile_pic_updated_at :datetime
user_id :integer
first_name :varchar
last_name :varchar
occupation :varchar
about_me :text
created_at :datetime
updated_at :datetime
profile_pic_file_name :varchar
profile_pic_content_type :varchar
profile_pic_file_size :integer
id :INTEGER
Subscription
id :INTEGER
subscriber_id :integer
subscribed_id :integer
created_at :datetime
updated_at :datetime


ApplicationController
get_event
only_event_owner
_layout
CommentsController
create
destroy
index
new
show
update
_layout
comment_params
DeviseController
_prefixes
subscriptions
unknown_action!
clean_up_passwords
devise_i18n_options
devise_mapping
find_message
navigational_formats
require_no_authentication
resource
resource=
resource_class
resource_name
resource_params
respond_with_navigational
scope_name
set_flash_message
set_flash_message!
set_minimum_password_length
signed_in_resource
successfully_sent?
translation_scope
assert_is_devise_resource!
_layout
EventsController
update
delete_event_pic
destroy
edit
index
new
show
subscribe
subscribers
subscriptions
unsubscribe
create
_layout
event_params
ManageEventsController
delete_tweet
destroy
index
update
_layout
PagesController
home
_layout
ProfilesController
create
delete_profile_pic
edit
new
update
_layout
only_current_user
profile_params
SearchEventsController
index
_layout
TweetsController
delete
destroy
index
_layout
tweet_params
UsersContr
index
show
_layout
Devise::ConfirmationsController
create
new
show
after_confirmation_path_for
after_resending_confirmation_instructions_path_for
translation_scope
_layout
Devise::OmniauthCallbacksController
failure
passthru
after_omniauth_failure_path_for
failed_strategy
failure_message
translation_scope
_layout
Devise::PasswordsController
create
edit
new
update
after_resetting_password_path_for
after_sending_reset_password_instructions_path_for
assert_reset_token_passed
translation_scope
unlockable?
_layout
Devise::RegistrationsController
cancel
create
destroy
edit
new
update
update_resource
after_inactive_sign_up_path_for
after_sign_up_path_for
after_update_path_for
authenticate_scope!
build_resource
sign_up
sign_up_params
translation_scope
update_needs_confirmation?
account_update_params
_layout
Devise::SessionsController
create
destroy
new
auth_options
serialize_options
sign_in_params
translation_scope
_layout
all_signed_out?
respond_to_on_destroy
verify_signed_out_user
Devise::UnlocksController
create
new
show
after_sending_unlock_instructions_path_for
after_unlock_path_for
translation_scope
_layout
OmniauthCallbacksController
all
twitter
_layout
Users::PasswordsController
after_resetting_password_path_for
_layout
Users::RegistrationsController
after_sign_up_path_for
after_update_path_for
_layout
Users::SessionsController
after_sign_in_path_for
_layout

Designing object-oriented iteration diagrams and class diagrams for Ruby on Rails, which uses the Model-View-Controller paradigm instead of object-oriented, is quite the challenge. You have the Model, which essentially acts as our object that contains various attributes and some methods, you have the view, which displays the UI of the application, and you have the Controller, which acts as a bridge between the Model and the View to determine how the two will interact on certain actions.

We designed our Iteration Sequence Diagrams using our Models, and the actions between them use various methods from the Controllers and Models because they both work hand-in-hand. For example, the use case “Add Comments” starts with the User Model who will click a button that triggers the new action in the Comments Controller. The Comment Model will then return the “comment new view” where the User can create their comment. Once the User selects the add comment button, the create action is triggered in the Comments Controller. The Comment Model then receives the parameters from the comment new view and saves the comment in the ActiveRecord Model, which is the database. The database is SQLite3 in development and PostgreSQL in production.

There needed to be two class diagrams for Ruby on Rails, one for the Model and one for the Controller. In the Model Class Diagram, every Model is inherited from the ActiveRecord Model. Every Model is also displayed with how it interacts with each Model. For example, a Comment has one Event and an Event has many Comments. Each Model also displays its attributes which are stored in ActiveRecord, or the database. In the Controller Class Diagram, each Controller is inherited from the Application Controller, which controls all other controllers. Each Model has its own Controller as seen in the Controller Class Diagram. Each Controller

contains various methods, public being on top, followed by protected, and then private. For example, the Users Controller has two methods, Index and Show, which display all Users in a view, or just displays the specific User in the view.

The GRASP patterns used here are Controller and Creator. The Controller pattern is pretty much identical to how a Controller works in Ruby on Rails, so that is automatically included. The Controller pattern is used to explain how to connect the UI layer (View) with the application logic (Model). For example, the Comments Controller connects the Add Comment button with the Comment model which contains all the logic on what to do with a new comment. The Creator pattern is also used because we are creating new instances of an object, such as a Comment. The Creator in our case is hard to describe in an object-oriented case, but it is the ActiveRecord Model which creates a new record in the DB, representing a "new object." When we want to create a new comment, we are actually creating a new record in the database, and then showing all of those instances in the database in the view. The Model in our case is represented by the Creator pattern.