1 Introduction

Build an application that allows users to create public and private Markdown-based wikis.

Overview and Purpose

In this project you'll create a CRUD application using Ruby on Rails.

Objectives

After this project, you should be able to:

- Create a basic user scheme for a Ruby on Rails application.
- Give the users of a Ruby on Rails application the ability to sign up for your application using the Devise gem.
- Understand how to create CRUD routes and resources in a Rails Application.
- Explain the difference between authentication and authorization.
- Integrate Stripe third party API callouts.
- Integrate Redcarpet Markdown rendering gem.

Use Case

Wikis are a great way to collaborate on community-sourced content. Whether the wiki is for a hobby or work-related project, you will build an app that lets users create their own wikis and share them publicly or privately with other collaborators.

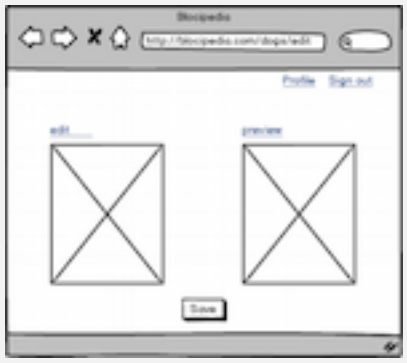
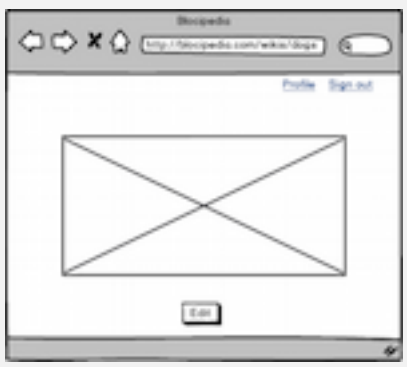
User Stories

User Story	Difficulty Rating
As a user, I want to **sign up** for a free account by providing a user name, password and email	2
As a user, I want to **sign in and out** of Blocipedia	2
As a user with a standard account, I want to **create, read, update, and delete** public wikis	3
As a developer, I want to offer three user roles: admin, standard, or premium	4
As a developer, I want to **seed** the development database automatically with users and wikis	1
As a user, I want to **upgrade** my account from a free to a paid plan	4
As a premium user, I want to **create** private wikis	3
As a user, I want to **edit** wikis using Markdown syntax	2
As a premium user, I want to **add** and **remove** collaborators for my private wikis	3

Before you begin working on user stories, complete this project's **Getting Started guide**. Later user stories often rely on the completion of the former, therefore, work on them in the order prescribed.

Wireframe Examples

These wireframes are meant to suggest a design, not dictate one.

Wireframe	Description
	A sign up page.
	An index.
	Edit the Wiki page.
	Show the Wiki.
	Add collaborators.

How would you rate this checkpoint and assignment? ☺ ☹

1. Introduction

Assignment | Discussion | Submission

2 User Sign Up

> As a user, I want to **sign up** for a free account by providing a user name, password and email
> **Difficulty Rating**: 2

Incorporate Devise

Use the **Devise** gem for authentication. Blocipedia's authentication system should allow users to sign up and send emails for account confirmation. Refer to the **Devise Resource** and the **Devise Getting Started Guide** for examples on implementing user sign up.

Test Your Code

- Sign a new user up. Do you receive a confirmation email?
- What happens if you attempt to sign up with an invalid email?
- What happens if you attempt to sign up with a duplicate email?

How would you rate this checkpoint and assignment? ☺ ☹

3 User Sign in and Out

> As a user, I want to **sign in and out** of Blocipedia
> **Difficulty Rating**: 2

Now that users can sign up for Blocipedia, you want to give them a way to sign in and out of the app. Refer to our **Devise Resource** and the **Devise Getting Started Guide** for examples on implementing user sign in/out with Devise.

Test Your Code

- Sign into Blocipedia, does the top navigation change to indicate you are signed in?
- Sign out of Blocipedia, does the top navigation change to indicate you are signed out?
- What happens if you attempt to reset your password?

How would you rate this checkpoint and assignment? ☺ ☹

4 Wiki CRUD

> As a user with a standard account, I want to **create, read, update, and delete** public wikis
> **Difficulty Rating**: 3

Create the Model

Create the wiki model. For guidance, you can refer to the **Models Checkpoint**. Start by generating the Wiki model:

Terminal

```
$ rails g model Wiki title:string body:text private:boolean user:references:index
```

The `wikis_controller` will replace the `posts_controller` in Blocipedia.

Test Your Code

- Are you able to create new public wikis?
- Are you able to update public wikis?
- Are you able to delete public wikis?

How would you rate this checkpoint and assignment? ☺ ☹

4. Wiki CRUD

Assignment | Discussion | Submission

5 User Roles

> As a developer, I want to offer three user roles: admin, standard, or premium
> **Difficulty Rating**: 4

Incorporate Pundit

Use the **Pundit** gem for authorization. Users should have one of three roles: standard (free), premium, or admin. Refer to the **Pundit Readme** checkpoint for examples of using Pundit policies.

Default to Standard

Users should default to the standard role when they are first created. There are **several ways** to implement default values. Use the `after_initialize` callback approach to implement default values for roles.

In Bloccit, a user either needed to be an admin, or the post creator to edit a post. In Blocipedia, users should be able to edit any public wiki. To allow this behavior, change the `update?` method in `application_policy.rb`:

app/policies/application_policy

```
def update?
  user.present?
end
```

For an extra challenge, implement user authorization from scratch. This **blog post** provides a great starting point.

Test Your Code

- Use the Rails console to create a new user. Are they given the default role?
- Are you able to edit wikis created by another user?

How would you rate this checkpoint and assignment? ☺ ☹

5. User Roles

Assignment | Discussion | Submission

6 Seeding Data

> As a developer, I want to **seed** the development database automatically with users and wikis
> **Difficulty Rating**: 1

Seed With Faker

Now that user and wiki models are established, you can seed data into the development database. Use the **Faker** gem to generate fake 1 data for **users and wikis**.

Return to this user story as you continue to build Blocipedia, updating `seeds.rb` to reflect the changes you make in the app.

Test Your Code

- Rebuild your database, is it seeded with the data you specified?

How would you rate this checkpoint and assignment? ☺ ☹

7 Upgrading an Account

> As a user, I want to **upgrade** my account from a free to a paid plan
> **Difficulty Rating**: 4

Determine a User Flow

> A *user flow* is the path a user follows to complete a task in the app.

Here is a hypothetical user flow for Blocipedia:

1. The user signs up for a free plan.
2. The user upgrades their free plan and is prompted to pay with Stripe.
3. The user's role is changed from standard to premium.
4. The user is able to create private wikis.

Can you think of alternative flows that result in the user's role changing from standard to premium?*

Incorporate Stripe

Use **Stripe** to charge users before switching their account role from standard to premium. Make the price of the premium user upgrade 15 dollars. Implement Stripe using the **Stripe Integration Resource**.

You could also upgrade a user before you charge them via Stripe. Why would you want to charge users before you upgrade them?* Review the **Happy Path** article on Wikipedia.

Downgrade Back to Standard

You should also allow a user to downgrade their premium account to a standard account.

What complications does this add to your application?* Design and implement a user flow for this scenario.

Test Your Code

- Upgrade an existing user account. Was your Stripe account charged? Was the user role changed?
- Downgrade a premium user. Was your Stripe account charged? Was their role changed?

How would you rate this checkpoint and assignment? ☺ ☹

Stripe Integration

Stripe's popularity is due largely to its developer-friendliness. Stripe removes much of the messiness from the payment integration process. It verifies credit card information for you, shielding you from the actual data so that your users are secure. It charges the cards for you, and routes the money wherever you want (**with some restrictions**). But payment gateways are complicated by nature, and even with Stripe, stringing together a payment system for your app can be difficult to conceptualize.

So let's take a look at a basic Stripe integration. There are countless ways to integrate Stripe into your app; we'll cover a simpler method here.

Getting Started

As with many Rails common problems, gems are a great place to start. The **Stripe Gem** greatly simplifies working with the Stripe API. Add the gem to your Gemfile:

Gemfile

```
+ gem 'stripe'
```

And don't forget to:

Terminal

```
bundle install
```

Once you've installed Stripe, head on over to **the site** to register for

an account. Once signed in, you'll have access to an account dashboard with information like your balance, purchase history, customers, and more.

Most importantly for right now, this new account gives you access to personalized API keys, accessible through the account drop-down in the upper right.

> The toggle in the upper left of your account page allows you to switch between "Live" and "Test" views. This will prove very useful throughout the development process.

Armed with API keys, we're ready to head back to the app. At a high level, here's the thinking:

- First things first, we need to hook our app up to our Stripe account.
- Once we've made it through configuration, we move onto the actual charging. We want to treat charges like any old resource, so we should create a **ChargesController** that that can initialize new **Charge** objects, and then "create" charges (sending them to Stripe through its API).
- When Stripe receives the Charge information from your app, with your API keys, it verifies the card and, if all looks good, runs the charge.
- If something goes wrong (bad card information, or luck), the API will return an error message, and, just as with an error in a name field, the user can be redirected to try again.
- Once the charge has passed through, you can see it on your account page.

Configuration

The first step in this process is configuration. The cleanest way to do this is in a separate initialization file. Let's call it `config/initializers/stripe.rb`:

config/initializers/stripe.rb

```
+ # Store the environment variables on the
Rails.configuration object
+ Rails.configuration.stripe = {
+   publishable_key: ENV['STRIPE_PUBLISHABLE_KEY'],
+   secret_key: ENV['STRIPE_SECRET_KEY']
+ }
+
+ # Set our app-stored secret key with Stripe
+ Stripe.api_key =
Rails.configuration.stripe[:secret_key]
```

Nice and easy, right? Don't forget that you have to **add these environment variables, locally and on production,** before you use them.

Congrats! Once you've set your env variables, you're all done with configuration.

"Creating" Charges

Onward to step two - the creation of the **ChargesController**, where charges will be initialized and created.

Let's start with creation. The `create` action of the controller will receive params and turn them into a charge object to send to Stripe. In this example, we won't actually be storing this charge object in our database, because Stripe takes care of the bookkeeping for us.

The params are:

- `stripeToken` - A token, unique to the charge, that contains the encrypted credit card information to be sent to Stripe.
- `amount` - The payment amount *in pennies*. (Pennies are less of a pain to deal with than fractional money amounts. Trust us. And Stripe.)

So, that action could look something like this:

app/controllers/charges_controller.rb

```
+ def create
+   # Creates a Stripe Customer object, for associating
+   # with the charge
+   customer = Stripe::Customer.create(
+     email: current_user.email,
+     card: params[:stripeToken]
+   )
+
+   # Where the real magic happens
+   charge = Stripe::Charge.create(
+     customer: customer.id, # Note -- this is NOT the
user_id in your app
+     amount: Amount.default,
+     description: "BigMoney Membership - #
{current_user.email}",
+     currency: 'usd'
+   )
+
+   flash[:notice] = "Thanks for all the money, #
{current_user.email}! Feel free to pay me again."
+   redirect_to user_path(current_user) # or wherever
+
+   # Stripe will send back CardErrors, with friendly
messages
+   # when something goes wrong.
+   # This `rescue block` catches and displays those
errors.
+   rescue Stripe::CardError => e
+     flash[:alert] = e.message
+     redirect_to new_charge_path
+ end
```

> We'll leave it to you to create a `Amount` class and its `default` class method, which should return the number you wish to charge, in pennies. One such value might be `10_00`, for ten dollars.

Before we get to the `new` action, one quick detour into views. Here's a basic **charges/new.html.erb**:

charges/new.html.erb

```
+ <%= form_tag charges_path do %>
+   <h4>Click the button!</h4>
+   <script class='stripe-button'
src="https://checkout.stripe.com/checkout.js" data-key="
<%= @stripe_btn_data[:key] %>" data-amount="<%=
@stripe_btn_data[:amount] %>" data-description="<%=
@stripe_btn_data[:description] %>" ></script>
+ <% end %>
```

Confused by that variable? We're declaring it in the controller, just to clean up the view:

app/controllers/charges_controller.rb

```
+ def new
+   @stripe_btn_data = {
+     key: "#{
Rails.configuration.stripe[:publishable_key] }",
+     description: "BigMoney Membership - #
{current_user.name}",
+     amount: Amount.default
+   }
+ end
```

To get this all to work, add a new `resources` route to point to the `ChargesController` actions:

config/routes.rb

```
+   resources :charges, only: [:new, :create]
```

OK. So it all works, maybe. But how do you test this all out? Stripe provides several **test card numbers**, everyone's favorite of which is `4242 4242 4242 4242`. So to test out charging, navigate to the `charges/new` page on local, click the button, fill in that card number and a fake CVC and (future) expiration date, and submit the charge.

It should show up shortly on your Stripe dashboard, in the Test tab.

8 Private Wikis

> As a premium user, I want to **create** private wikis
> **Difficulty Rating**: 3

Implement Privacy Controls

Premium and admin users should be able to create new private wikis and make public wikis private. Check the user's role before allowing them to edit a wiki's private attribute:

app/views/wikis/_form.html.erb

```
<% if current_user.admin? || current_user.premium? %>
  <div class="form-group">
    <%= f.label :private, class: 'checkbox' do %>
      <%= f.check_box :private %> Private wiki
    <% end %>
  </div>
<% end %>
```

Downgrade Private Wikis

Since Blocipedia allows premium users to downgrade their accounts, what should happen to their private wikis?* Build a user flow for this scenario which prompts the user with a reminder that their private wikis will become public if they downgrade their account.

Test Your Code

- As a premium user, create a private wiki. Sign in with a standard user. Do you see the private wiki in the wikis index?
- As a premium user, create a private wiki. Downgrade your account to standard. Verify that their private wikis are made public.

How would you rate this checkpoint and assignment? 

8. Private Wikis

Assignment | Discussion | Submission

9 Markdown

> As a user, I want to **edit** wikis using Markdown syntax
> **Difficulty Rating**: 2

Incorporate Redcarpet

Use the **Redcarpet** gem to parse Markdown syntax.

Test Your Code

- View a wiki page built with Markdown, does it render the Markdown properly?

How would you rate this checkpoint and assignment? ☺ ☹

10 Wiki Collaborators

> As a premium user, I want to **add** and **remove** collaborators for my private wikis
> **Difficulty Rating**: 3

Modify the Edit Page

Users will add and remove collaborators from a private wiki via its edit page.

Create a Collaborator Model

Model this new relationship between wikis and users by creating a collaborator model. Read through our resource on **Has Many Through** relationships to get a sense for how you might want to relate private wikis with users through collaborators.

Private wikis should only be visible to admins, the wiki creator, or wiki collaborators. Use Pundit's `scope` to restrict which wikis appear on the index page. To do so, add an inner `Scope` class to `wiki_policy.rb`:

app/policies/wiki_policy.rb

```
  class WikiPolicy < ApplicationPolicy

  ...

+   class Scope
+     attr_reader :user, :scope
+
+     def initialize(user, scope)
+       @user = user
+       @scope = scope
+     end
+
+     def resolve
+       wikis = []
+       if user.role == 'admin'
+         wikis = scope.all # if the user is an admin, show them all the wikis
+       elsif user.role == 'premium'
+         all_wikis = scope.all
+         all_wikis.each do |wiki|
+           if wiki.public? || wiki.owner == user || wiki.collaborators.include?(user)
+             wikis << wiki # if the user is premium, only show them public wikis, or t
+           end
+         end
+       else # this is the lowly standard user
+         all_wikis = scope.all
+         wikis = []
+         all_wikis.each do |wiki|
+           if wiki.public? || wiki.collaborators.include?(user)
+             wikis << wiki # only show standard users public wikis and private wikis t
+           end
+         end
+       end
+       wikis # return the wikis array we've built up
+     end
+   end
  end
```

Use the scope in the `wikis_controller.rb` to display only the appropriate wikis:

app/controllers/wikis_controller

```
-   def index
-     @wikis = Wiki.all
-   end

+   def index
+     @wikis = policy_scope(Wiki)
+   end
```

Test Your Code

- As a premium user, add a standard user as the collaborator to a private wiki. Can you add the user multiple times? Sign in as the standard user. Do you see the private wiki in the wikis index? Can you edit the private wiki?
- As a premium user, remove a collaborator from a private wiki. Sign in as the ex-collaborator. Do you see the private wiki in the wikis index? Can you edit the private wiki?

How would you rate this checkpoint and assignment? ☺ ☹

Has Many Through

In Bloccit, we dealt primarily with relatively simple data relationships; a user `has_many` posts, which `belong_to` both `topics` and `users`, etc. Real-world data relationships are often more complicated. One of the common data relationships you'll find is referred to in the Rails community as the **Has Many Through** (**HMT**) relationship.

HMT in Bloccit

Let's think about **Has Many Through** in the context of our Bloccit app, where we actually used this relationship without naming it.

In Bloccit, users could "favorite" posts they liked. In this way, a user could "have many" favorite posts, even though none of those posts point directly to the user. As you may recall, we stored this relationship in a third model named `Favorite`. Each favorite had a `user_id` and a `post_id`, and if we wanted to find all the "favorited posts" that a user had, we could write:

Console

```
  favorites = Favorite.where(user_id: @user.id)
  favorited_posts = Post.where(id:
favorites.pluck(:post_id))
```

> This exact syntax is new. To get a better sense of it, take a look at the extremely useful `pluck` method and the use of `where` **with an array**.

This is a **Has Many Through** relationship, because users *have many*

posts *through* favorites. The favorites table is known as a "join table", because its purpose is to *join* the users and posts table in a unique way.

Another perspective

We could also think of the User-Favorite-Post relationship as a "**Has and Belongs to Many**" (**HABTM**) relationship. Users have many favorited posts, as we've already shown, but posts also *have many* "favoriting users". We could refer to all the users who "belong to" a given post - in terms of having favorited it - and all the posts which "belong to" a user who has favorited many posts.

> **HMT** and **HABTM** seem interchangeable. What's the difference?

In Rails, a slight distinction is made between **HABTM** and bi-directional **HMT** relationships. **Has Many Through** relationships emphasize the joining model, in this case `Favorite`. The join table in a **HMT** relationship has characteristics apart from its joining functions. For example, if users could favorite posts to varying degrees, so that `Favorite` had a `rating` attribute, we could refer to users having posts *through* favorites.

> As coded in Bloccit, however, the User-Favorite-Post relationship can be thought of as a **Has and Belongs to Many** relationship, because Favorite has no functionality or attributes aside from `user_id` and `post_id`.

Another example

Let's step outside of Bloccit to explore HMT further. We'll create a data model for an application that manages movie theater information. Let's think about our specifications:

- We should be able to list the movies showing at a given theater.
- We should be able to list every theater for a given movie.

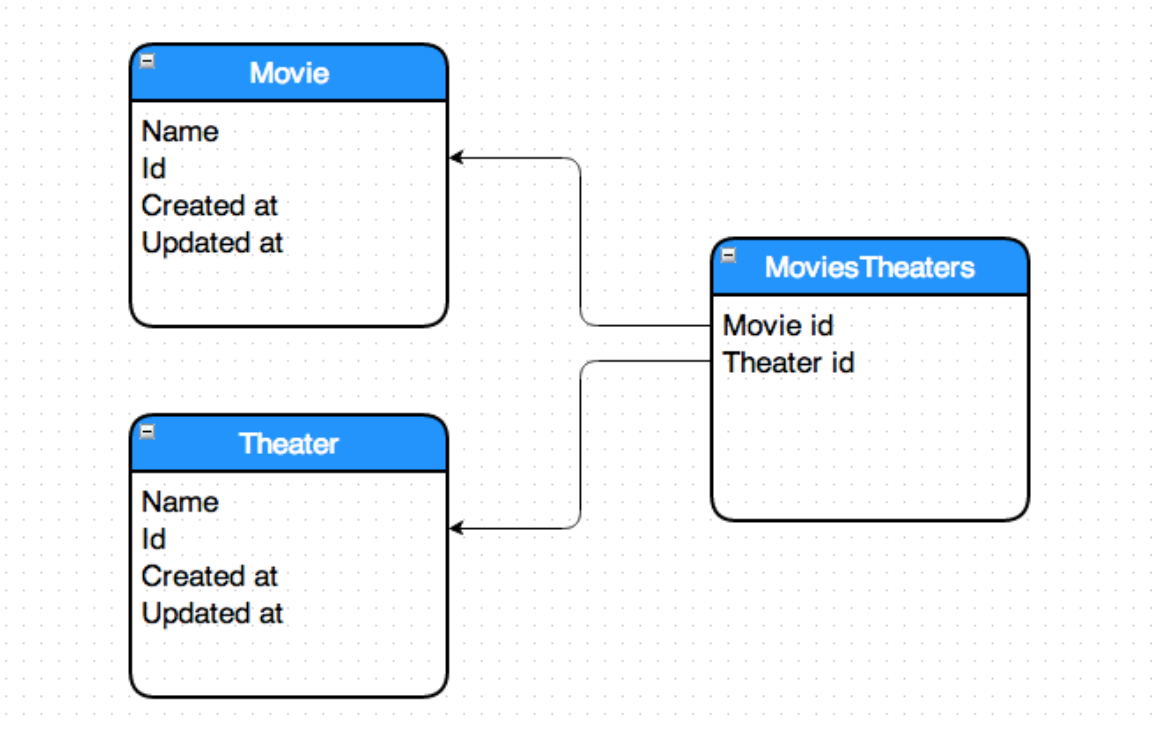
We'll obviously want a `Theater` model and a `Movie` model. Let's

think about how we'd relate these models to meet our criteria. A theater `has_many` movies, but a movie doesn't `belong_to` *a* theater, because it can show at *many* theaters. Instead, movies also *belong to many* theaters (some may say movies *have many* theaters).

These two criteria point us towards a **Has and Belongs to Many** relationship: Theaters have and belong to many movies; movies have and belong to many theaters.

> Think about how we might create this relationship in the database.

The solution is a join table. If you're having trouble figuring out what it should be named, that's alright. Join tables of these sorts often specify unintuitive, intangible relationships, and the Rails convention is just to name them as the combination of the two (pluralized) table names: `movies_theaters`.



Our `movies_theaters` table is a classic HABTM join table, and, as the meaningless table name suggests, it doesn't need any additional attributes. Let's create the migration to make these tables. The Rails Guide to **Association Basics** provides a good start (using some new syntax we'll discuss):

Terminal

```
$ rails g migration CreateMoviesAndTheaters
```

> We didn't specify attributes and types in the command itself. Rather we'll manually specify the model details in the migration file. In that file, we replace the `t.belongs_to` in the above example with `t.references`. **The two are identical**, so we'll stick with what we've used before.

db/migrate/20140701134132/create_movies_and_theaters.rb

```
+ class CreateMoviesAndTheaters < ActiveRecord::Migration
+   def change
+     create_table :movies do |t|
+       t.string :name
+       t.timestamps
+     end
+
+     create_table :theaters do |t|
+       t.string :name
+       t.timestamps
+     end
+
+     create_table :movies_theaters, id: false do |t|
+       t.references :movies
+       t.references :theaters
+     end
+   end
+ end
```

> The `id: false` `create_table` **option** specifies that this table is *not* a full-fledged model. We won't have a `.rb` file, and we won't be able to look up `movies_theaters` by `id`.

The `id: false` and `t.references` options are helpful, but removing the id and timestamps isn't necessarily a good idea. What if, down the line, we want to know when a theater added a movie, or we decide that this join table should have some additional attributes and be searchable by id?

Better safe than sorry. Let's change the migration, and give the join table a better name, turning the relationship into a **Has Many Through**:

> If you already ran the migration, you can roll it back with `rake db:rollback`.

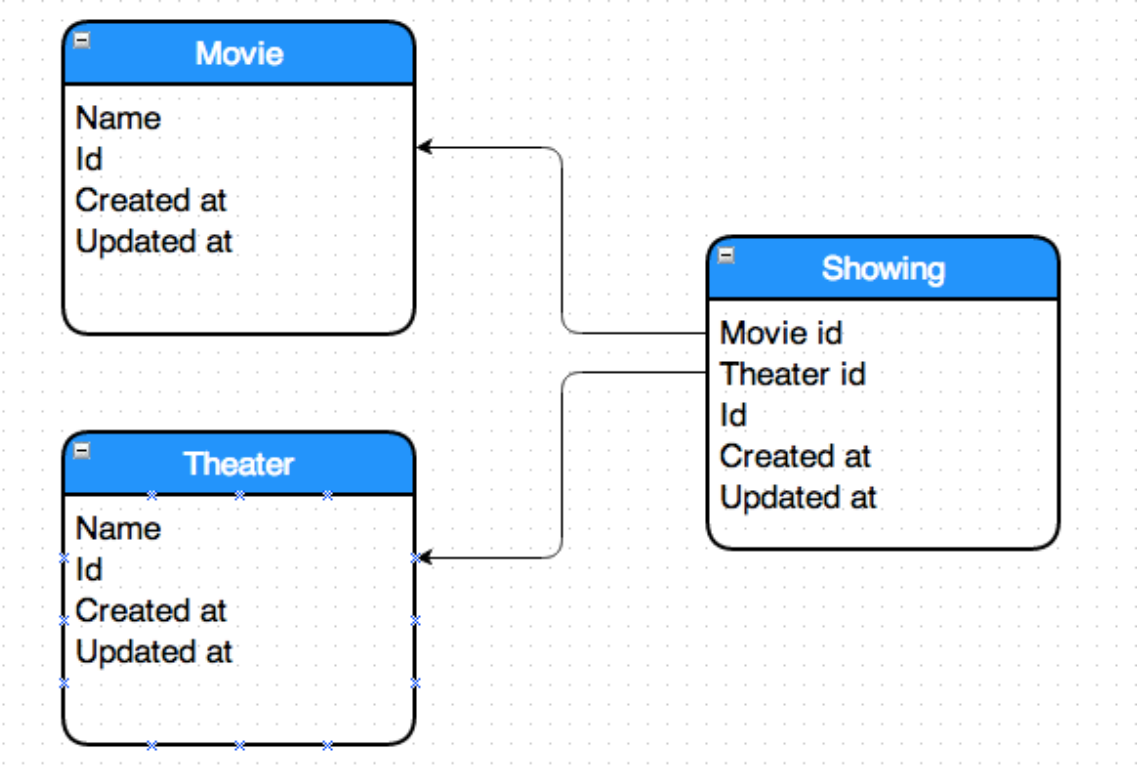
db/migrate/20140701134132/create_movies_and_theaters.rb

```
   class CreateMoviesAndTheaters < ActiveRecord::Migration
     def change
       create_table :movies do |t|
         t.string :name
         t.timestamps
       end

       create_table :theaters do |t|
         t.string :name
         t.timestamps
       end

-      create_table :movies_theaters, id: false do |t|
-        t.references :movies
-        t.references :theaters
-      end
+      create_table :showings do |t|
+        t.integer :movie_id
+        t.integer :theater_id
+        t.timestamps
+      end
+
+      add_index :movies, :id, unique: true
+      add_index :theaters, :id, unique: true
+      add_index :showings, :id, unique: true
+      add_index :showings, :movie_id
+      add_index :showings, :theater_id
     end
   end
```

> We added indexes to make searching by id faster for all three models. We also switched `t.references` to `t.integer` to be more explicit and avoid Rails auto-creation of relationship methods.



What is the value of this change? There are several:

- We've wedded our model to a real-world concept, making it significantly easier to understand.
- We've improved our opportunity for tracking data history. We can now tell when a showing was added to a theater (or "for a movie").

> When in doubt, err on the side of collecting and keeping as much data as possible. You can't get it back if you never tracked it.

- We've made it easier to explore and debug the model in the console. You can't directly query **HABTM** models using ActiveRecord.
- We've allowed our join table to have separate functionality. We'll use this shortly.

Run the migration with `rake db:migrate`

The model side

Because we generated a straightforward Rails `migration`, as opposed to a `model` migration, Rails will not have generated our model files. Let's write those now, starting with `Movie`:

app/models/movie.rb

```
+ class Movie < ActiveRecord::Base
+   def showings
+     Showing.where(movie_id: id)
+   end
+
+   def theaters
+     Theater.where( id: showings.pluck(:theater_id) )
+   end
+ end
```

Let's move onto `Theater`:

app/models/theater.rb

```
+ class Theater < ActiveRecord::Base
+   def showings
+     Showing.where(theater_id: id)
+   end
+
+   def movies
+     Movie.where( id: showings.pluck(:movie_id) )
+   end
+ end
```

Finally, let's add a `Showing` file:

app/models/showing.rb

```
+ class Showing < ActiveRecord::Base
+   def self.movies
+     Movie.where( id: pluck(:movie_id) )
+   end
+
+   def self.theaters
+     Theater.where( id: pluck(:theater_id) )
+   end
+
+   def theater
+     Theater.find(theater_id)
+   end
+
+   def movie
+     Movie.find(movie_id)
+   end
+ end
```

This `showing.rb` file allows us to refactor our `movie.rb` and `theater.rb` files:

app/models/theater.rb

```
    def movies
-     Movie.where( id: showings.pluck(:movie_id) )
+     showings.movies
    end
```

app/models/movie.rb

```
    def theaters
-     Theater.where( id: showings.pluck(:theater_id) )
+     showings.theaters
    end
```

The key concept here is that movies and theaters can `delegate` the calculation of `theaters` or `movies` to their showings, rather than re-defining those methods. Active Record has a nifty `delegate` method that allows us to do this explicitly. Let's use it to refactor again:

> The **delegate** method is extremely useful, and takes an options

> hash that can change delegation behavior. Be careful overusing it, however, as it can make searching for specific method calls difficult.

app/models/movie.rb

```
  class Movie < ActiveRecord::Base

+   delegate :theaters, to: :showings

    def showings
      Showing.where(movie_id: id)
    end
-
-   def theaters
-     showings.theaters
-   end
  end
```

Let's move onto `Theater`:

app/models/theater.rb

```
  class Theater < ActiveRecord::Base

+   delegate :movies, to: :showings

    def showings
      Showing.where(theater_id: id)
    end
-
-   def movies
-     showings.movies
-   end
  end
```

As you may have guessed, there's a final refactor we've been working towards; the use of the Active Record association methods. We *could* say that a theater `has_many` showings and then `delegate :movies, to: :showings`, which would bring our model down to two lines. But Rails makes life even easier. Using Rails associations correctly, we can refactor our three model files to the following (final)

code:

app/models/movie.rb

```
class Movie < ActiveRecord::Base
  has_many :showings
  has_many :theaters, through: :showings
end
```

app/models/theater.rb

```
class Theater < ActiveRecord::Base
  has_many :showings
  has_many :movies, through: :showings
end
```

app/models/showing.rb

```
class Showing < ActiveRecord::Base
  belongs_to :theater
  belongs_to :movie
end
```

> We could have skipped the intermediary steps and gone straight to the above code. However, the steps make the relationship clearer.

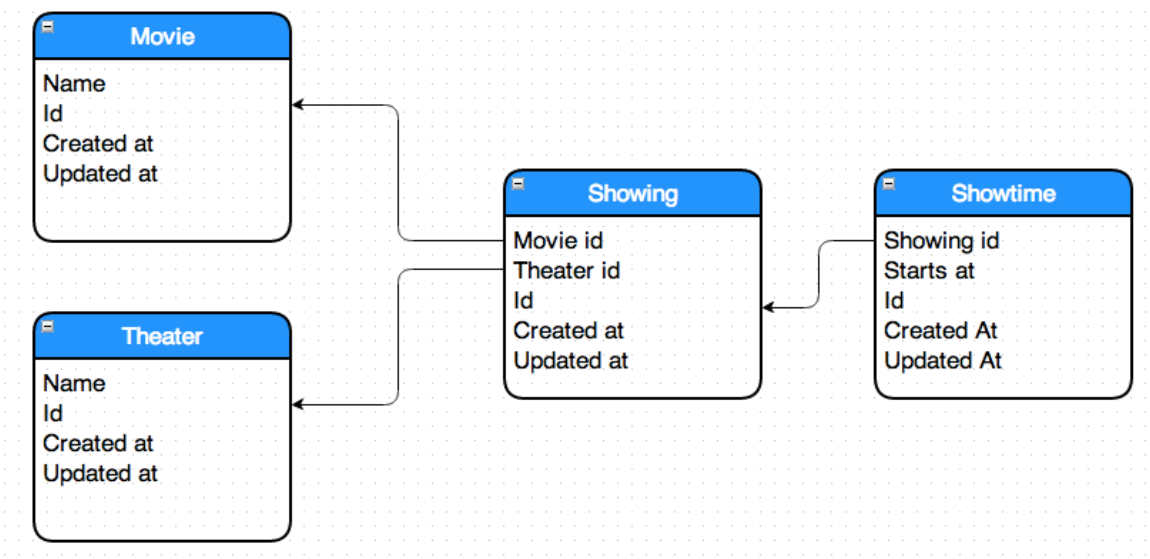
HMT vs HABTM

Earlier in this resource, we opted for **Has Many Through** over **Has and Belongs to Many**. We did this because it turned `Showing` into a full-fledged model, which has a number of advantages, not the least of which is increased flexibility.

Let's say we decide to build a concept of "showtimes" into the application, so that people can use our app to buy tickets to movies.

This showtime information should be stored in relation to a particular showing of a movie at a theater. In the **HABTM** model, *this would not have been possible* without altering the `MoviesTheaters` model or adding another largely redundant join model. Each row of

`movies_theaters` would have had no `id` for associations.



With **HMT**, this is significantly easier. We can generate a `Showtime` model, which `belongs_to` the `Showing` model (which, in turn, `has_many :showtimes`). In a way, showtime `belongs_to` a movie and a theater, through its `showing`. Rails does *not*, however, support the `belongs_to :movie, through: showing` syntax. Instead, there are two easy solutions. The first, slighly unintuitive, solution is to say that Showtime `has_one :movie, through: :showing`. The other is to use `delegate` again, explicitly saying that the `movie` for a given `showtime` can be found through its `showing`:

app/models/showtime.rb

```
  delegate :movie, :theater, to: :showing
```

The two solutions are effectively interchangeable. You can use whichever you prefer.

> How would you define the relationship between Movie and Showtime? Why can't it be referred to as a **HABTM** relationship? There is more than one reason.

The generated relationship methods are cool, but it can be easy to get carried away with them. When you find yourself creating all the possible permutations of a given model relationship, remember the **YAGNI** principle: "You aren't gonna need it."

A good example of potential overkill would be setting up a relationship between Movie and Showtime allowing you to call `movie.showtimes` directly. This is neat, but it represents behavior without much

application in the real world, and the odds are that *you aren't gonna need it*. Rather than creating "code bloat" by building a ton of functionality early, wait until it's absolutely necessary. When you do, use TDD to fulfill the need as specifically as possible. Overkill solutions can lead to headaches down the line.

11 Extra Credit

The goal of extra credit is to push you towards self-reliance, so outside of a brief description, we don't provide additional resources. If you finish your project early, challenge yourself to complete these additional user stories:

User Story	Difficulty Rating
As a developer, I want to generate Blocipedia's views using **HAML** instead of ERB	3
As a user, I want my wiki to have **readable URLs**	3
As a user, I want to see a **preview** of my Markdown as I edit it	4

How would you rate this checkpoint and assignment? ☺ ☹