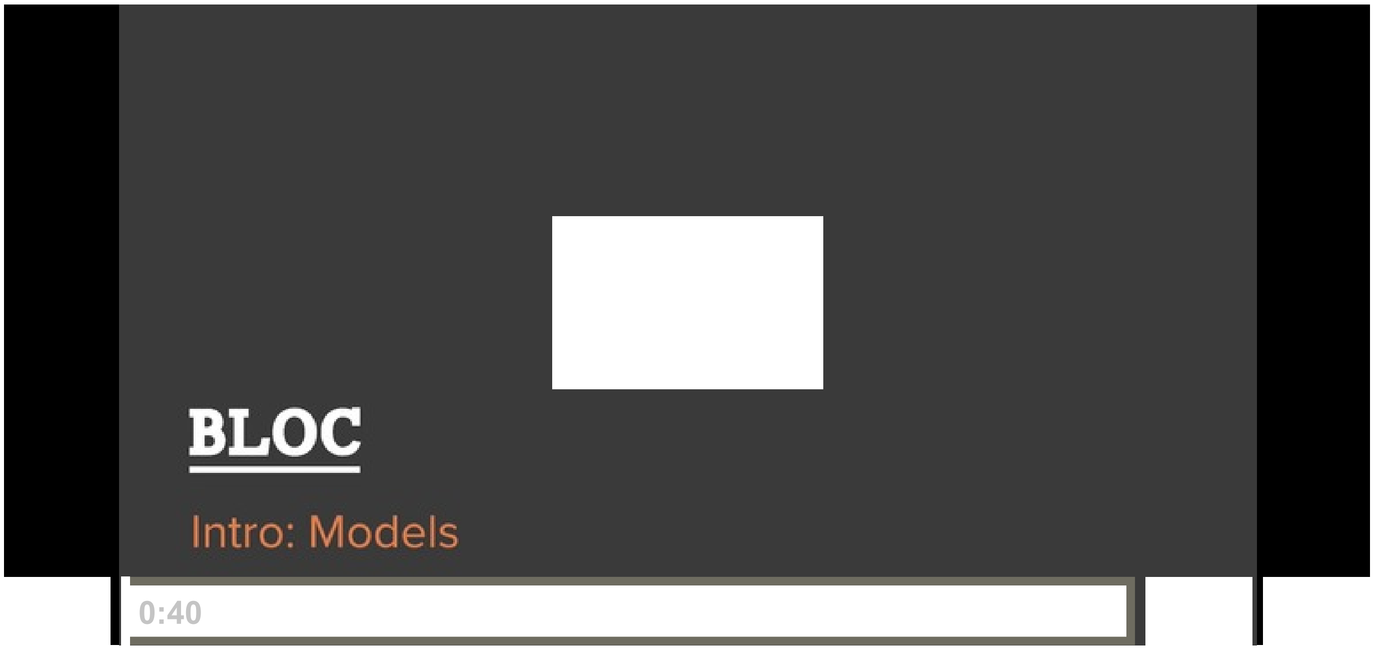
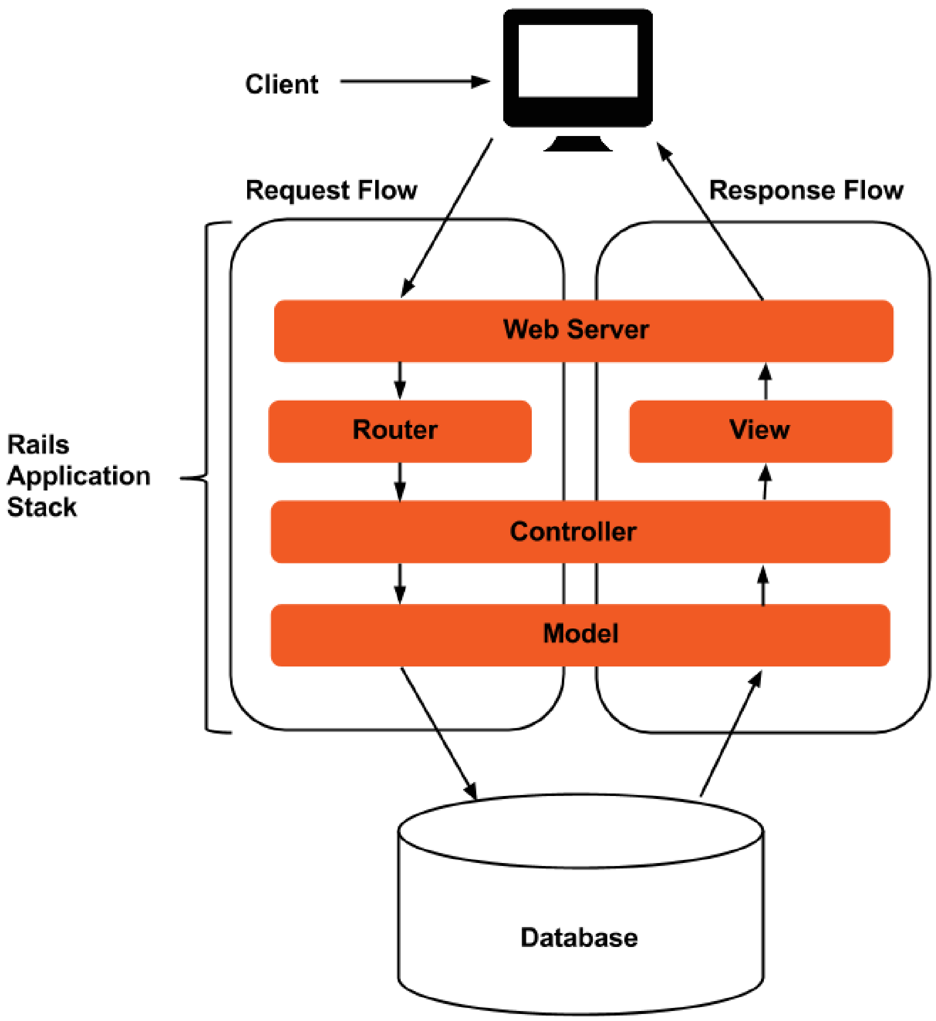
28 Rails: Models

> People come up to me all the time and say 'you should be a model', or 'you look just like a model,' or 'maybe you should try to be a man who models.' And I always have to laugh because I'm so good looking. Of course I'm a model.
>
> — **Derek Zoolander**

Models



Bloccit users will need to be able to *post* information and *comment* on those posts. Posts and comments will need to persist; that is, they'll need to be saved to a database so users can interact with them across sessions. When we need to persist data, we should immediately think about data models. Data models, or more simply "models", are the "M" in MVC architecture. Recall the diagram of MVC architecture to consider a model's place with respect to controllers and views:



Models are a programmatic representation of a table in a database. Models are also Ruby classes, similar to the classes we programmed earlier in the roadmap. In other words, the `Post` model will handle data pertaining to an instance of the `Post` class.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

`Post`

The first model we'll create is `Post`. Users should have the ability to submit posts to Bloccit with titles and descriptions, so the `Post` model and its corresponding database table will need two attributes: `title` and `body`.

> An attribute is synonomous with database table column. You can also imagine a database table

> as a tab on a spreadsheet. Attributes like "title" and "body" would be column headers on that spreadsheet tab.

Use a generator to create `Post` and its corresponding spec:

Terminal

```
$ rails generate model Post title:string body:text
      invoke  active_record
   identical    db/migrate/20150606010447_create_posts.rb
   identical    app/models/post.rb
      invoke    rspec
      create      spec/models/post_spec.rb
```

We used a generator to create a model named "Post" with two attributes: `title` and `body`.

- The `title` attribute is a string data type, because we expect it to be short. That is, we would probably not want to allow post titles with hundreds of characters.
- The `body` attribute is a text data type, because we expect a post's body to be verbose. It's possible that a user will need the ability to write hundreds of characters to provide context for a post.

The programmatic representation of the Post model was created by the generator above, in several files:

- `post.rb` is a Ruby class which represents the Post model. This class will handle the logic and define the behavior for posts.
- `post_spec.rb` is the test spec for the Post class.
- `20150606010447_create_posts.rb` is the database migration file. A migration file defines the action taken on the database for a given model. An application will have many migration files, and comprehensively they serve as a set of instructions for building a database. We'll explore migration files in detail later.

Use `cat` to see the contents of the spec file:

Terminal

```
$ cat spec/models/post_spec.rb
 require 'rails_helper'

 RSpec.describe Post, type: :model do
   pending "add some examples to (or delete) #{__FILE__}"
 end
```

This is the template for a simple spec. We'll use TDD to define the behavior for `Post`. Add the following tests:

spec/models/post_spec.rb

```
   require 'rails_helper'

   RSpec.describe Post, type: :model do
-    pending "add some examples to (or delete) #{__FILE__}"
   # #1
+    let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
+
+    describe "attributes" do
   # #2
+      it "responds to title" do
+        expect(post).to respond_to(:title)
+      end
   # #3
+      it "responds to body" do
+        expect(post).to respond_to(:body)
+      end
+    end
   end
```

- At **#1**, using the `let` method, we create a new instance of the Post class, and name it `post`. **let** dynamically defines a method (in this case, `post`), and, upon first call *within a spec* (the `it` block), computes and stores the returned value.
- At **#2**, we test whether `post` has an attribute named `title`. This tests whether `post` will return a non-nil value when `post.title` is called.
- At **#3**, we apply a similar test to `body`.

Use `cat` to see the contents of `post.rb`, which was also created with a basic template:

Terminal

```
$ cat app/models/post.rb
class Post < ActiveRecord::Base
end
```

When the generator created this template, it made the Post class **inherit** from **ActiveRecord::Base**. Because we used a model generator, Rails assumed that we wanted our class to be used as a **model**. ActiveRecord::Base essentially handles interaction with the database and allows us to persist data through our class. Run the spec:

Terminal

```
$ rspec spec/models/post_spec.rb
```

You will see a verbose error, but focus on the first line:

Terminal

```
schema.rb doesn't exist yet. Run `rake db:migrate` to create it, then try again.
```

RSpec reported that `schema.rb` doesn't exist. `schema.rb` is a file located in the `db` directory that represents an application's complete database architecture; the tables it uses and how those tables relate to each other. We don't have `schema.rb` because we have not yet created the database or any tables. The generator created the migration file, but we haven't executed that file yet. We'll do that now:

Terminal

```
$ rake db:migrate
== 20150606010447 CreatePosts: migrating ======================================
-- create_table(:posts)
   -> 0.0016s
== 20150606010447 CreatePosts: migrated (0.0017s) =============================
```

> Rake is a Ruby build command. It allows us to execute administrative tasks for our application. To see a complete list of rake tasks, type `rake --tasks` from the command line.

`rake db:migrate` created a new table named "posts". Let's review the migration file, which is the only file in the `db/migrate` directory (its name begins with a timestamp, and so will differ from the one below):

db/migrate/20150606010447_create_posts.rb

```
$ cat db/migrate/20150606010447_create_posts.rb
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body

      t.timestamps null: false
    end
  end
end
```

The migration is written in Ruby. The migration file is actually a class named `CreatePosts`. When we run the migration, the `change` method calls the `create_table` method. `create_table` takes a block that specifies the attributes we want our table to possess.

> Rails automatically adds timestamp attributes named `created_at` and `updated_at` to the migration. We'll discuss these attributes in depth later.

Run the tests in `post_spec.rb` again:

Terminal

```
$ rspec spec/models/post_spec.rb
..

Finished in 0.00817 seconds (files took 1.67 seconds to load)
2 examples, 0 failures
```

Our tests passed, so we know that the Post model has the attributes we expected.

`Comment`

The `Comment` model needs one attribute - `body` - and a reference to `Post`. Let's create the spec, model, and migration files with the model generator:

Terminal

```
$ rails generate model Comment body:text post:references
      invoke  active_record
      create    db/migrate/20150608215948_create_comments.rb
      create    app/models/comment.rb
      invoke    rspec
      create      spec/models/comment_spec.rb
```

Open `comment_spec.rb` and add the following test:

spec/models/comment_spec.rb

```
  RSpec.describe Comment do
-   pending "add some examples to (or delete) #{__FILE__}"
+   let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
+   let(:comment) { Comment.create!(body: 'Comment Body', post: post) }
+
+   describe "attributes" do
+     it "responds to body" do
+       expect(comment).to respond_to(:body)
+     end
+   end
  end
```

Review `comment.rb`:

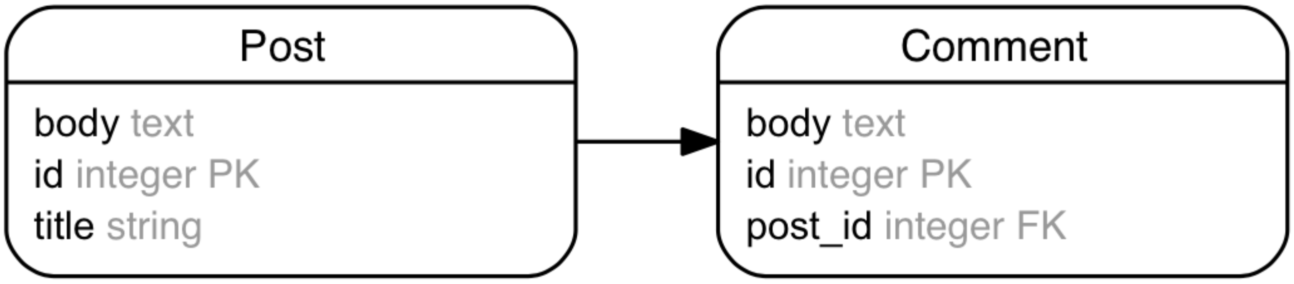
app/models/comment.rb

```
$ cat app/models/comment.rb
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

We say that *"a comment belongs to a post"* or, conversely, *"a post has many comments."* Both of these phrases relate to a data concept known as "relationships" or "associations."

Each model instance in a Rails app automatically gets an `id` attribute to uniquely identify it. Each `post` will have a unique `id`, as will each `comment`. To make a comment belong to a post, we need to provide the post `id` to the `comment`. This is done using a **foreign key**.

A foreign key is the `id` of one model, used as an attribute in another model, in order to look up the relationship. In the Post/Comment example, this means that the `Comment` model needs to have an attribute named `post_id`. The `post_id` attribute exists so that a `comment` can belong to a `post` (the post specified by its `post_id`). To allow many comments to belong to one `post`, you'd have multiple comment records with the same `post_id`. The diagram below illustrates how the post's `id` attribute relates to a comment's `post_id` attribute:



> Rails is an opinionated framework that enforces many conventions by design. The foreign key naming convention of `post_id` was enforced when you ran the model generator. The `post_id` attribute was automatically created in the `Comment` model when you generated it with the `post:references` argument.

Review the `create_comments` migration in the `db/migrate` directory and add a foreign key:

db/migrate/20140624203804_create_comments.rb

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.text :body
# #4
      t.references :post, index: true, foreign_key: true

      t.timestamps null: false
    end
  end
end
```

At **#4**, the `index` on the references line tells the database to index the `post_id` column, so it can be searched efficiently. This is always a good idea for foreign keys, and is added automatically when you generate with the `references` argument.

In reviewing the comment and post migrations, we see that the `create_table` method takes a `Symbol` argument which represents the table name, and a block argument that contains the details to be added to the table. This is one of the many reasons why Rails developers can code so efficiently - rather than creating the tables manually, and making sure all the attributes are set properly, we can rely on Rails' model generator to handle this mundane work.

Since we have a new migration file, we shall once again run the migrations, adding the comments table to the database:

Terminal

```
$ rake db:migrate
== 20150608215948 CreateComments: migrating ===================================
-- create_table(:comments)
   -> 0.0021s
== 20150608215948 CreateComments: migrated (0.0022s) ==========================
```

The results above tell us that the tables and attributes have been created successfully.

Run `comment_spec.rb`:

Terminal

```
$ rspec spec/models/comment_spec.rb
.

Finished in 0.01325 seconds (files took 1.78 seconds to load)
1 example, 0 failures
```

Git does not create empty directories by default. As a consequence, Rails automatically generates a blank `.keep` file in important directories that start as empty in a new application. Our `app/models` directory has one such file. Now that we've added two files to `app/models`, we should remove it:

Terminal

```
$ rm app/models/.keep
```

You've successfully created two database tables and associated them using a foreign key. In the next section we'll update the models to reflect the attributes and associations we just created in the console.

Updating `Post`

Remember that when we use a model generator, the resulting model inherits from a class named `ActiveRecord::Base` by default. This inheritance pattern provides methods that the model will need in order to interact with tables in the database. Our comment class already relates to our post class, thanks to the model generator, but the post class does not yet relate to the comment class. Let's create that relation:

app/models/post.rb

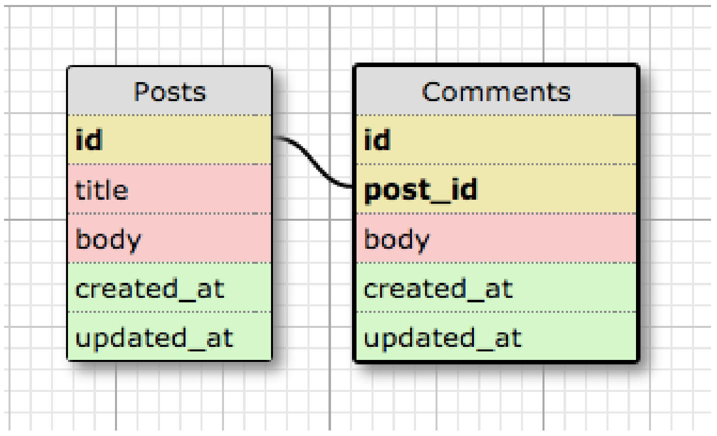
```
  class Post < ActiveRecord::Base
+   has_many :comments
  end
```

The `has_many` method allows a post instance to have many comments related to it, and also provides methods that allow us to retrieve comments that belong to a post.

> This dynamic generation is similar to the way `attr_accessor` generates 'getter' and 'setter' methods for instance variables. We'll explore the precise methods created by `has_many` in the next checkpoint.

Visualizing the Database

It can be helpful to think about databases visually: What tables does the database have? What attributes do the tables have? How are the tables related? Here's a sample visualization for the two tables we've created in our application:



When we say that a comment belongs to a post, we mean that the comment stores that post's unique identifier in an attribute. The visualization makes this obvious.

In the proceeding video, we use a popular Ruby IDE, **RubyMine**, to visualize the database for demonstrative purposes only.

> You do not need to download RubyMine during your Bloc course. Feel free to discuss it with your mentor and of course download it if you both feel like it will be helpful, but it is not a requirement for Bloc.

Git

Commit your checkpoint work in Git. See **Git Checkpoint Workflow: After Each Checkpoint** for details. Then deploy to Heroku.

Recap

Concept	Description
Active Record Models	Active Record Models are the layer responsible for representing business data and logic. They facilitate the creation and use of objects whose data requires persistent storage to the database.
Generating Models	The `rails generate` command uses templates to create models, controllers, mailers, and more. When used to generate a model, it creates the Ruby class, test spec, and Active Record database migration.
Active Record Migrations	Active Record Migrations allow you to evolve your database schema over time. They use Ruby so that you don't have to write SQL by hand.

28. Rails: Models

Assignment | **Discussion** | **Submission**

Create a new Git feature branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

Users may desire the ability to ask questions in Bloccit. The Questions model will be similar to the Posts model, but Questions should also have a `resolved` attribute that allows an administrator to mark the question as resolved.

1. Create a new model named `Question`. It should have `title:string`, `body:text`, and `resolved:boolean` attributes.
2. Create another new model named `Answer`. It should reference `Question` and

have a `body` attribute.

3. Write the specs for the `Question` and `Answer` models. The specs should ensure that the model attributes can be called as methods, as we did in the checkpoint with `Post` and `Comment` specs.
4. Update `Question` so that it `has_many :answers`.
5. Confirm that `Answer` `belongs_to :question`, in the Rails console.

Commit your assignment in Git. See **Git Checkpoint Workflow: After Each Assignment** for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the assignment. If you struggle to complete the assignment, submit your best effort to your mentor *before watching a solution video*.

Models Solution

assignment completed

?