Active Record

Introduction

Presumably you're here to learn web development (otherwise... you may be in the wrong place...). Whether your goal is to be able to produce your own website or to begin a career as a developer, the most important skillset to take away from all this is the ability to think logically and to break down a problem into its component pieces. Then you can address those pieces one at a time. It's the essence of engineering.

Probably the most important way that logical thinking is required when building a website is in setting up your data model properly. Data is the foundation of almost all major web applications, from a simple blog site to Facebook's massively complex web of data. Having an obscure or overly complex data model can cripple you when you try to grow and make your life as a developer exceedingly painful. If you're working with the wrong tools, something "simple" like asking to display all the comments a user has made on another user's web posts can take up far too many brain and CPU cycles to accomplish.

If data is the most important piece of a web application, then how Rails handles data should be very interesting to you. Luckily, this is one of the most significant reasons that Rails has performed so well compared with the options available just a few years ago. Active Record is the interface that Rails gives you between the database and your application. It lets you structure your data models for your users, blog posts, comments, followers, etc. in a logical and nearly plain-English way. If it seems complicated (which it will at points), just imagine life before Active Record.

Having a solid understanding of Active Record will make the rest of Rails seem simple by comparison. Recall from several lessons ago that the Model in MVC is the part that does all the heavy lifting. In this lesson, we'll cover all the basics of working with models, from setting them up to building simple associations between them. As usual, this explanation is meant to be a high level overview and the readings will provide real depth. The more advanced topics will be covered in some of the coming lessons.

Points to Ponder

Look through these now and then use them to test yourself after doing the assignment

•What is an ORM?

•Why is Active Record more useful than just using SQL?

•What are the two steps required to make a new row in your database table with ActiveRecord?

•What are "generators" in Rails?

What is an ORM?

What is Active Record anyway? Recall that Rails is actually seven Ruby gems that work harmoniously together. Active Record is, to put it inelegantly, the gem that takes care of all the database stuff. It's known as an "ORM".

ORM stands for Object-Relational-Mapping. It basically means that Active Record takes data which is stored in a database table using rows and columns, which needs to be modified or retrieved by writing SQL statements (if you're using a SQL database), and it lets you interact with that data as though it was a normal Ruby object.

So if I want to get an array containing a listing of all the users, instead of writing code to initiate a connection to the database, then doing some sort of `SELECT * FROM users` query, and converting those results into an array, I can just type `User.all` and Active Record gives me that array filled with User objects that I can play with as I'd like. Wow!

Even more impressive, it doesn't really matter which type of database you're using (as long as you've set up the `config/database.yml` file properly), Active Record smooths out all the differences between those databases for you so you don't have to think about it. You focus on writing code for your application, and Active Record will think about the nitty gritty details of connecting you to your database. It also means that if you switch from one database to another, you don't actually need to change any major application code, just some configuration files. Sounds logical, right?

Rails Models

That's a step ahead of ourselves, though, because first it makes sense to think about what the relationship is between Rails and a database anyway. It's actually pretty straightforward -- you want to store information about your users, so you create a database table called `users`. You want to be able to access that data from your application, so you create a model called `User`, which is really just a Ruby file which inherits from Active Record and thus gets to use all the handy methods I was alluding to earlier like `all` and `find` and `create`. One table corresponds with one model which inherits from Active Record.

30 Seconds About Working With Models

Very briefly, Active Record lets you create a Ruby object that represents a row in one of your database tables, like a `User`. To create a new User is a two-step process: First, you'll need to do a `User.new` and might pass it a hash full of its attributes like

```
u = User.new(name: "Sven", email: "sven@theodinproject.com")
```

If you don't pass a hash, you'll need to manually add the attributes by setting them like with any other Ruby object: `u.name = "Sven"`. The second step is to actually save that model instance into the database. Until now, it's just been sitting in memory and evaporates if you don't do anything with it. To save, simply call `u.save`. You can run both steps at once using the `#create` method:

```
u = User.create(name: "Sven", email: "sven@theodinproject.com")
```

This saves you time, but, as you'll see later, you'll sometimes want to separate them in your application.

Your Assignment

That was really just a teaser about what Active Record can do. In the reading below, you'll learn about how to specifically interact with Active Record in your models.

1. I'm assuming that you've already read and followed along with the example application that was created in the Getting Started with Rails intro section of the Rails Guides. If you haven't, do that first!

2. Read the Active Record Basics section of the Rails Guides.

- We'll go more into Migrations and Validations in the next section and in the lesson on Callbacks later in the course.

- Model files in Rails live in the `app/models` folder and are just normal .rb files. The key points are that the file and the class name is named after the table in your database (but singular), and that class inherits from ActiveRecord::Base to get its super powers.

Migrations

When You Need Them

Imagine you're staring at a blank computer screen and you need to start your new Rails project. What's the first thing you do? You type `$ rails new MyProjectName` then `cd` into that directory... Then what?

Figure out the data models that you'll need to use for the first iteration of your site and start getting them set up. For our purposes, we'll just assume all you need is the ubiquitous User model to keep track of all the dozens of users who will be on your site someday (just kidding, you'll hit it big). After you've actually created the database in the first place (using `$ rake db:create`), to create that model you'll need to do two steps:

1.Create a model file in `app/models` which is set up like you just learned above.

2.Create a database table called "users" that has the appropriate columns. This is done using a migration file and then running the migration.

The best part is that Rails knows that you want to do this and has given you a handy shortcut for doing so: the `$ rails generate model YourModelNameHere` command. When you type it in, you will see in the Terminal output which files are being created. Don't worry about any specs or test files that also get created, the important ones are the model file and the migration file. Rails has lots of these handy generators which don't do much except create new files in the right spots of your application for you. The output looks something like:

```
invoke  active_record
create    db/migrate/20131223154310_create_testmodels.rb
create    app/models/testmodel.rb
invoke    rspec
create      spec/models/testmodel_spec.rb
```

The model file that the generator creates is just a bare-bones model file in the `app/models` directory (which you could easily have created yourself). The other main file is the migration file in the `db/migrations`folder, which starts with a complicated looking timestamp like `20130924230504_create_users.rb`.

If you dive into that file, you'll see that there's not much in it except another bare-bones ruby class that inherits from `ActiveRecord::Migration`and some timestamps[*]. The timestamps just create `created_at` and `updated_at` columns for you so you can track when your database records were created or modified. These two columns are just helpful enough that they are included as standard practice.

If you want to only create the database migration file (without the Model or any of the test files), just use `$ rails generate migration NameYourMigration`. You'll end up using this one more once you've got things up and running since you'll probably be modifying your data table instead of creating a new one. There's a syntax for specifying additional parameters when you call this (which you'll see in the reading), but there's no need to remember that syntax because you can also manually go in and edit the migration file yourself.

[*]: Unless you passed the Rails generator the column names you wanted, in which case they would show up automatically in the migration fields. Generators let you pass in arguments to do even more for you.

What Are They?

So what's a migration? A migration is basically a script that tells Rails how you want to set up or change a database. It's the other part of Active Record magic that allows you to avoid manually going in and writing SQL code to create your database table. You just specify the correct Ruby method (like the aptly named `create_table`) and its parameters and you're almost good to go.

Migrations are just a script, so how do you tell Rails to run that script and actually execute the code to create your table and update your database's schema? By using the `$ rake db:migrate` command, which runs any migrations that haven't yet been run. Rails knows this because it keeps track of which migrations have been run (using timestamps) behind the scenes. When you run that command, Rails will execute the proper SQL code to set up your database table and you can go back to actually building the website.

Why is this useful? Obviously it lets you set up your database using user-friendly Ruby code instead of SQL, but it's more than that. Over time, you'll build up a bunch of these migration files. If you decide that you want to blow away your database and start from scratch, you can do that easily and then rerun the migrations. If you decide to deploy to the web, you will run those same migrations and the production database will be there waiting for you... even if it's a different type of database! Again, Active Record does the heavy lifting for you here so you can focus on building your website.

The most immediately useful feature of migrations is when you've screwed something up because they're (usually) reversible. Let's say you just migrated to create a new database column but forgot a column to store the user's email... oops! You can just type `$ rake db:rollback` and the last series of migrations that you ran will be reversed and you're back to where you were. Then you just edit the file, rerun the migrations, and move on with your life.

This introduces the last nuance of migrations that we'll talk about here -- reversibility. For each method that you use in the migration, you want to specify how to reverse it if you have to. The reverse of adding a table is dropping that table, of adding a column is removing the column and so on. Many methods have a really obvious reverse case, so you don't need to explicitly state it and can set up the whole migration file using the `change` method. But some of them do not, so you will need to separately specify `up` and `down` methods. You'll read more about that in the assignment.

A final note, you never want to rollback migrations unless you've screwed something up. In situations where you have a legitimate case for removing a column (because you no longer need it for any purpose), you actually create a new migration that removes that column using the `remove_column` method. It preserves the database. Once you get advanced with this stuff, you can build a database just using the schema file... You're not there yet :)

How Much Database Stuff do I Need to Know?

Migrations don't involve writing SQL, but you do need to understand enough about databases to know how you want yours structured! Which columns do you want? Which ones should be indexed (and why)? Should you set a default value? What data type will be stored in your column... a string or text?

These are great questions, and you should feel comfortable asking them even if you aren't totally sure about the answers. If you have no idea what I'm talking about, you'll need to go back and read up on basic databases in the previous lesson.

Your Assignment

1.Read the Migrations chapter of Rails Guides.

•Don't worry about 3.6-3.8.

•Just skim section 7.

•Seeds (section 8) are useful and you'll be using them later. It saves you a lot of work, especially when you're learning and will end up blowing away your database and starting over a lot.

Basic Validations

Imagine you've got your database up and running and want to make sure that the data people are sending to your database is good data. For instance, to create an account on your site, a user needs to enter both a username and an email address. How do you enforce this?

There are three levels of validations that you can enforce, each more strict and secure than the previous. At the topmost level, you can write code using JavaScript in your browser that detects if someone has filled out the form properly and will prompt them to finish it before moving on. We will learn more about that in the JavaScript course. The advantage here is that it is almost immediate so has great user experience. The problem with this is that JavaScript is easy to circumvent and the user could easily submit a malicious or faulty request.

The second layer of enforcement for your validations of user data (which you should never trust) is to do so at the server level. This means writing code in your Rails application (specifically in the model that you are trying to save an instance of, e.g. User) that examines user inputs, checks them versus the constraints you set up, and returns errors if there are any.

This is more secure than JavaScript but has the disadvantage of taking a full round-trip HTTP request to your application in order to check it. Model validations are generally pretty effective and that's what we'll focus on here.

Another problem occurs when your application has scaled up to the point where you are running multiple instances of it on multiple servers that all talk to the same central database. Let's say you want to make sure a username is unique... what happens if two users almost simultaneously submit the same username and it is received by two separate concurrent instances of your application? When each instance of your application checks with the database to see if the username is unique, both times it looks okay so they both go ahead and save the model... oops! That may not sound plausible, but how about in rapidly occurring automated transactions? These "race conditions" are very real.

So the only way to truly enforce constraints is on the database level, since your single database is the sole arbiter of what is unique and valid in this world. You can use extra parameters passed to some of the now-familiar migration methods like `add_index` to say `add_index :users, :username, unique: true`, which enforces in the most secure way that the column is unique. Again, though, most of your validations can be implemented in your Rails application's models.

Your Assignment

1.Read the Rails Guides Validations chapter

•Section 2 on helpers can be skimmed -- these help you get more specific with your validations and you'll run into them later

•You can skim section 6 about custom validators

•Section 8 will likely only be interesting if you've seen ERB in rails views before... we'll get there.

Basic Associations

In the databases sections, you learned about how a relational database like sqlite3 or PostgreSQL lets you link two tables together using their primary keys (called a foreign key in the specific table that is referencing another one). It's the real power of relational databases that they let you leverage these, well, relationships. Active Record takes that feature and lets you use it in all kinds of useful ways. Do you want to get all of your first user's blog posts? Try `User.first.posts`. It's as simple as that.

That functionality doesn't come out of the box -- you need to tell Rails that posts actually belong to a user. On the database table level, this means that every row in the posts table will have column for `user_id`that tells you which user "owns" that post. The users table doesn't need to acknowledge the posts at all... after all, a single user can have an infinite number of posts. If we're interested in a user's posts, we just have to query the posts table for all posts that link back to that user's ID. Rails makes these relationships very easy to specify. What we just talked about is aptly named a "has many / belongs to" association (a User `has_many` Post objects associated with it and a Post `belongs_to` a single User).

Step one with understanding this stuff is just to think about which different types of relationships are possible. Remember, half the battle of setting up your application is understanding what your data relationships will look like, so give this some thought and keep at it when it gets confusing. If your mind is a bit fried right now, start back in the real world and don't think about it on a database level -- remember, all this stuff is our attempt to reflect the kinds of relationships that can occur in the real world.

The `has_many` / `belongs_to`, or a "one-to-many", relationship is pretty common, and usually easiest to think of in terms of actual objects... a Child can have many Marble objects, each of which belongs to that Child. But it also applies in slightly less intuitive cases, like where a single object `belongs_to` multiple other objects. An example would be a FranchiseLocation for a McDonalds, which `belongs_to` the Corporation Mcdonalds but might also `belongs_to` the City San Francisco.

It's clear that it should belong to its corporate parent, but why does it belong to a City too? It's often easier to think of it from the opposite perspective -- a City can certainly have many FranchiseLocation objects. As long as a FranchiseLocation can only be in a single city, it effectively "belongs_to" that city in the way that Rails describes it.

Another common relationship is the many-to-many relationship, which can also be called `has_and_belongs_to_many` in Rails terms. This often comes up in actual relationships -- a Human can have many favorite Dog objects, and each Dog object can have many favorite Human objects. In this case, how would you specify which Dog objects are your favorites? It actually requires you to create another table (a join table, or "through" table) that specifically keeps track of all those relationships. It's a bit wonky to understand when you're learning but it becomes second nature once you've been at it for a short while.

A key distinction here is that we're not talking about how many Post objects a User *currently* has or how many FranchiseLocation objects a City *currently* has, we're trying to model how many they *COULD* have over the entire lifetime of your application. Sure, if your User only has one Post now, you could hard code that post's ID in your user table. Then when he has another Post, you'd have to create another table column to fit that ID. And another. And another. Which doesn't make a lick of sense... so that's why we say the User `has_many :posts` and let the Posts table hardcode in the User's ID, since a post will only ever have one User associated with it (assuming you only have one author).

Pretty soon you'll start thinking of the world around you in terms of these relationships (if you don't take enough breaks). The real power of them comes when you actually need to use them -- when you want to retrieve data about all the objects that are associated with another. Do you want to see a list of all your Twitter followers? Do you want to count up all the classmates you had from high school who are living in the same city as you now? Do you want to see all the comments one of your users left on another user's timelines? All of these things are relatively simple and intuitive once you've actually set up the appropriate data relationships. So focus on understanding these relationships.

Your Assignment:

If you're a normal human, you're probably somewhere between "huh?" and "I hate you, stop teaching me stuff". Stick with it, the point here is to get you thinking of how to model relationships and give you exposure to them. The project will give you an opportunity to actually build what you've been learning and it should be a lot better once you've had that chance.

1. Read the beginning of the Rails Guides Associations Chapter, just up until section 2.7. Everything after that we can save for later... the important thing is that you've seen the relationships and how they're set up.

Conclusion

Active Record is the most powerful part of Rails and also one of the trickiest to get the hang of. You need to be able to translate the real world into database tables, which takes a bit of time to become familiar with. The most difficult concepts for new beginners are usually associations.

It's easiest to start thinking about concrete relationships in the real world and then turning them into Active Record associations. Once you're comfortable with which model `has_many` of which other model and who actually `belongs_to` the other, you can start modeling more abstract concepts like, say, event invitations and invitation acceptances, which aren't as straightforward as a Child and his marbles (the example we used above).

It's all about practice, so the projects from here on out will ask you to think through your model organization before getting started. Taking a few minutes to think through your relationships ahead of time is essential for getting started in the right direction when you begin writing code.