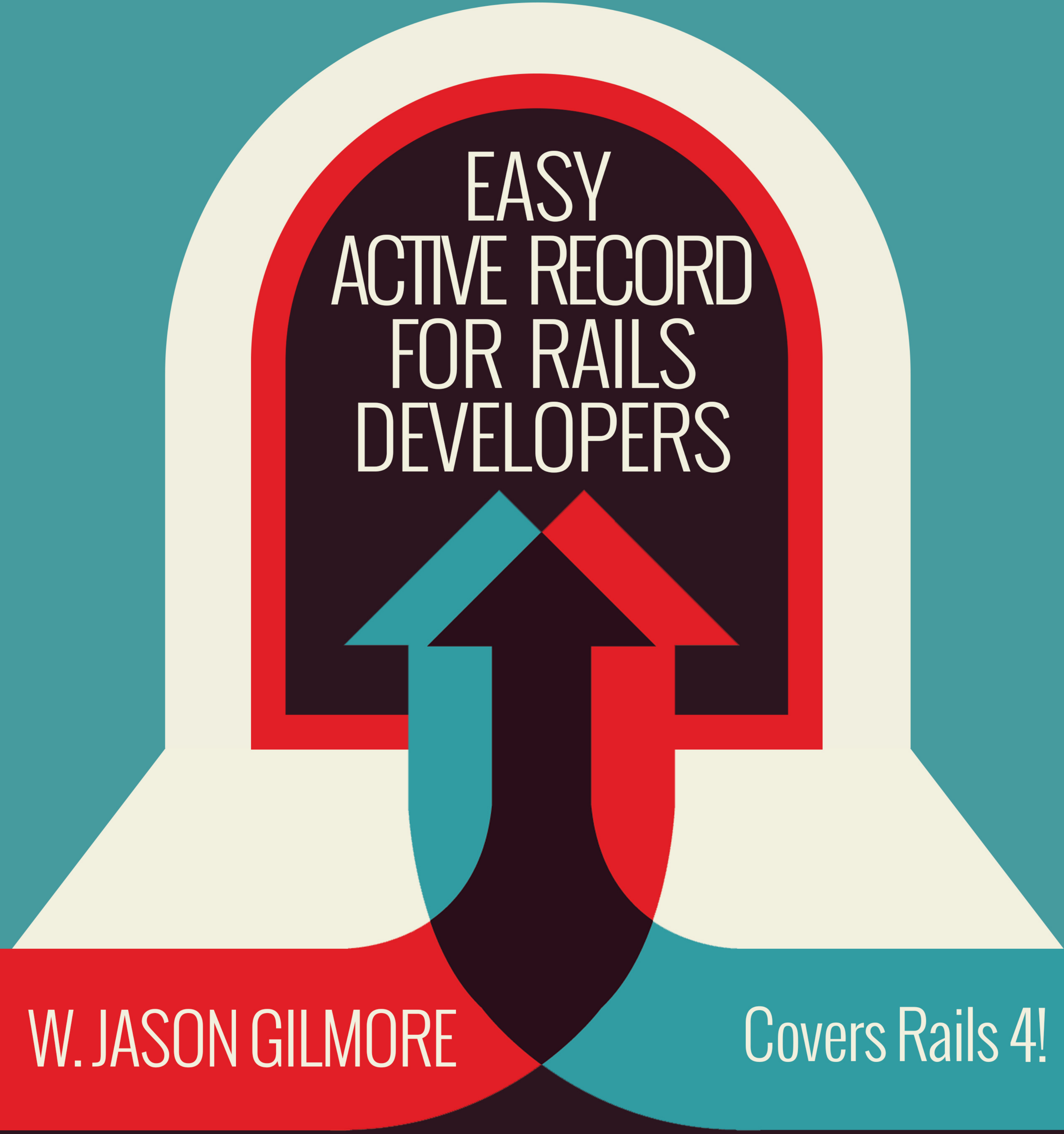
A Hands On Introduction Using
a Real-World Project



EASY ACTIVE RECORD FOR RAILS DEVELOPERS

W. JASON GILMORE

Covers Rails 4!

wjgilmore.com

Easy Active Record for Rails Developers

Master Rails Active Record and Have a Blast Doing It!

W. Jason Gilmore

This book is for sale at http://leanpub.com/easyactiverecord

This version was published on 2015-01-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For Carli, my bride today and forever.

Contents

Introduction

Even in an era when reportedly game-changing technologies seem to be released on a weekly basis, the vast majority of today's applications continue to manage critical data within the decidedly unhip relational database. That's because even in these heady days of rapid technological evolution, the relational database remains a capable, fast, and reliable storage solution. Yet building a database-driven web application in a maintainable, scalable and testable fashion can be an extraordinary challenge!

The most notable historical challenge has involved surmounting the so-called "impedance mismatch" in which the developer is tasked with writing the application in not one but *two* languages: the declarative SQL and an object-oriented language such as PHP, Ruby, or Python. Not only does the developer have to constantly sit mentally astride both languages, but he's also faced with the issue of figuring out how to effectively integrate everything without producing a giant ball of spaghetti code.

Fighting the impedance mismatch is only the beginning of one's troubles. Presumably the application's data schema will be in a constant state of evolution, particularly in the early stages of development. How will the desired schema changes be incorporated into the database? How can mistaken changes be reverted without undue side effects? How will the schema changes be efficiently migrated into your production environment? None of these questions have easy answers, yet your application's success, not to mention your sanity, will partly hinge on your ability to deftly handle such matters.

Additional burdens lie in the ability to effectively test the data-oriented components of your application. It is foolhardy to presume your code is perfect, yet attempting to manually test these features is an exercise in madness and ultimately one that over time will cease to occur. The only viable testing solution involves automation, yet exactly how one goes about implementing this sort of automation remains a mystery to many developers.

Fortunately, the programming community is an industrious lot, and strives to remove inefficiencies and complexities at every opportunity. As such, quite a bit of work has been put into overcoming the aforementioned challenges. One of the most successful such efforts to remove not only the many database-related obstacles faced by web developers, but additionally a whole host of other challenges associated with web application development, is the Ruby on Rails framework[1]. This book is devoted to helping you master Ruby on Rails' (henceforth referred to as *Rails*) powerful database-integration and management features.

[1] http://www.rubyonrails.org/

Introducing Active Record

My favorite programming book is undoubtedly Martin Fowler's "Patterns of Enterprise Application Architecture[2]". It is one of the few quality books devoted to explaining how powerful, maintainable software applications should be designed and developed. Much of this book is devoted to a survey of key *design patterns* (a generally reusable solution to a recurring software design problem), among them the *Active Record* pattern. Pulling out my trusty copy, I'll quote Fowler's definition:

> An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

According to this definition, the Active Record pattern removes the aforementioned impedance mismatch by wrapping the database tables in classes that serve as the conduit for data access. Further, developers are empowered to extend the class capabilities through the addition of domain logic. In doing so, not only does the Active Record pattern provide an easy way to carry out object-oriented CRUD (create, read, update, delete) database operations, but it also facilitates management of other behavioral aspects of the entity represented by the table (such as ensuring a user's name is not left blank, or tallying up all of the comments a user has left on your website).

So what does this mean in concrete terms? Let's consider a few real-world examples.

> ℹ Saving a Few Keystrokes
>
> From here on out when I mention the term *Active Record* I'll be referring to it in the context of the Ruby on Rails implementation saving the hassle of typing out *Rails Active Record* or some similarly repetitive variation.

Convenient Object-Oriented Syntax

This book's theme project is called ArcadeNomad[3], a location-based application identifying the locations of retro 1980's arcade games (ArcadeNomad is formally introduced in the later section, "Introducing the ArcadeNomad Theme Project"). Among ArcadeNomad's many requirements is the simple task of creating and retrieving arcades. Although in the chapters to come you'll be formally introduced to the Active Record syntax used to implement such features, this syntax is so intuitive that I wanted to at least offer a cursory example to get you excited about what's to come. Let's begin by creating a new location:

[2] http://www.amazon.com/gp/product/0321127420?&tag=wjgilmore

[3] http://arcadenomad.com

```
location = Location.new
location.name = "Dave & Buster's Hilliard"
location.street = "3665 Park Mill Run Dr"
location.city = "Hilliard"
location.state = State.find_by(abbreviation: "OH")
location.zip = "43026"
location.save
```

Don't fret over the details of this example right now; just understand we're creating a new instance of the `Location` model, assigning a name, street, city, state, and zip code, and then saving that instance to the database by inserting a new record into the `locations` table. This record can later be retrieved in a variety of fashions, including by name:

```
location = Location.find_by(name: "Dave & Buster's Hilliard")
```

Of course, effective use of such an approach requires you to designate the `name` attribute as unique in order to avoid the possibility of multiple similarly named records being retrieved. I'll show you how this is done in Chapter 1.

Easy Model Associations

You just witnessed a simple example involving the creation of a new database record of type `Location`. In this example each attribute is assigned using a string, except for the column representing the location's state of residence. In the case of the state of residence, the `Location` model and `State` model are related using a `belongs_to` association. This great feature ensures the location's associated state is properly *normalized*, thereby eliminating any possibility of naming inconsistencies and other issues that otherwise arise when inputting data in a free-form manner. This feature also offers an incredibly convenient syntax for traversing those associations. For instance suppose the `State` model identifies each state by its abbreviation and name, meaning the state of Ohio's abbreviation would be `OH` and its name would logically be `Ohio`. Now suppose we retrieve the aforementioned location named `Dave & Buster's Hilliard` and want to know the name of the location's state-of-residence. The syntax is incredibly easy:

```
location = Location.find_by(name: "Dave & Buster's Hilliard")
puts location.state.name
```

In Chapter 4 I'll introduce you to the many ways Rails allows you to associate models, and show you how to deftly traverse these associations.

Model Validations

You might be wondering what would happen if somebody attempted to create a new arcade location that lacked one or more attribute values, or used values deemed invalid. Consider the following example:

```
location = Location.new
location.name = ''
location.city = 'Hilliard'
location.state = State.find_by(abbreviation: 'OH')
location.zip = 'qwerty'
location.save
```

There is plenty wrong here; the location's `name` attribute is assigned a blank value, the `street` attribute is altogether missing, and the `zip` attribute is assigned a nonsensical value. While this example is perhaps melodramatic (although not if it depicted a malicious attempt to enter erroneous data), it does highlight a potential issue that left unchecked could sink the ArcadeNomad application. After all, users are unlikely to return if it's filled with errant and missing data.

Fortunately, Active Record offers a powerful feature called *Validations* that you can use to ensure your models will not accept missing or invalid data, and inform users accordingly when an attempt is made to insert undesirable data. For instance, you can ensure your `Location` model will deny any attempts to insert a blank `name` attribute by defining a presence validator:

```
validates :name, presence: true
```

Similarly you can ensure the `zip` field only accepts a five digit integer sequence (such as `44425` and `43016`) by combining multiple validators like this:

```
validates :zip, numericality: { only_integer: true }, length: { is: 5 }
```

In Chapter 2 I'll offer an extensive overview of Active Record's validation capabilities.

Domain Logic Management

One of ArcadeNomad's key features is *geolocation*, providing users with a convenient way to learn more about the arcades in their vicinity. To do so we'll need to perform a relatively complex calculation known as the Haversine formula[4]. Fortunately thanks to Ruby's rich gem ecosystem we'll rely on the Geocoder[5] gem to do the dirty work for us, but even so it always makes sense to encapsulate the gory details associated with identifying nearby arcades using the Geocoder gem within a model method rather than pollute the controllers with such computational logic.

With this method added to the `Location` model, retrieving for instance the locations within 10 miles of the `Dave & Buster's Hilliard` location is as easy as retrieving the desired record and then calling the `nearby` method, passing in the desired radius:

[4] http://en.wikipedia.org/wiki/Haversine_formula

[5] https://github.com/alexreisner/geocoder

```
location = Location.find_by(name: "Dave & Buster's Hilliard")
nearby_locations = location.nearby(10)
```

The `nearby_locations` variable would then typically be populated with an array of `Location` objects which you can then present to the user.

In Summary

The sort of intuitive syntax presented in these preceding examples is typical of Active Record, and in the following chapters we'll review plenty of additional snippets illustrating the power and convenience of this great Rails feature. Keep in mind that what you've seen so far is only but a taste of what you're about to learn. A few more of my favorite Active Record features include:

- *Intuitive Table Join Syntax*: Your project requirements will almost certainly exceed the illustrative but simplistic example presented earlier in this section highlighting Active Record's ability to retrieve associated data. Fortunately, this is but one sample of Active Record's ability to join multiple tables together. I'll talk about this matter in Chapter 4.
- *Sane Database Schema Management*: Thanks to a feature known as *migrations*, you will be able to manage the evolution of your project's database schema within the source repository just like any other valuable file. Furthermore, you can advance *and rollback* these changes using a convenient command line interface, the latter feature proving particularly useful when a mistake has been made and you need to revert those changes.
- *Easy Testing*: Automated testing is such a crucial part of the development process that we'll return to it repeatedly throughout this book. Thanks to the powerful Rails ecosystem, there are several fantastic testing gems that we'll employ to ensure the code is properly vetted.

About this Book

Unlike some of the other books I've written, some of which have topped out at over 800 pages and attempt to cover everything under the sun, this book focuses intently upon a single concept: mastering Rails' Active Record implementation. Therefore while relatively short, my goal is to provide you with everything which can be reasonably and practically discussed regarding this fascinating bit of technology. Further, this book is decidedly *not* a general introduction to Rails, therefore I'll presume you're already familiar with fundamental concepts such as how to create new projects, controllers, and views. If you're not familiar with these concepts, or require a refresher course, be sure to check out Michael Hartl's excellent online book, "Ruby on Rails Tutorial[6]".

Let's briefly review the seven chapters comprising this book.

[6] http://www.railstutorial.org/

⚠ Mind the Rails Version

While Rails 3 developers will find much of what's covered in this book both interesting and useful, the material has been written specifically with Rails 4 in mind. There are numerous reasons why you should upgrade to Rails 4 if you haven't already, among them performance, stability, and general framework improvements. I'll be careful to point out Rails 4-specific features when applicable, but Rails 3 readers should nonetheless be wary of version-specific issues.

Chapter 1. Introducing Models, Schemas and Migrations

In this opening chapter we'll get acquainted with many of the fundamental Active Record features you'll use throughout your project's lifetime. You'll learn how to create the models used to manage the application's business logic and provide convenient interfaces to the underlying database. You'll also learn how to use a great feature called *migrations* to create and evolve your database schema. The chapter concludes with a section explaining how to begin testing your models with RSpec[7] and factory_girl[8].

Chapter 2. Loading, Validating and Manipulating Data

In this chapter you'll learn how to easily load, or *seed*, your application with an initial data set, as well as create and manipulate data. Of course, because we'll want to eliminate all possibilities of incomplete or invalid data from being persisted to the database, you'll also learn how to enhance your application models using a variety of *validators*. Additionally, we'll build on the introductory model testing material presented in the previous chapter. Notably, you'll learn how to create tests for ensuring you've properly fortified your models with enough validators to ensure unwanted data can't slip through the cracks.

Chapter 3. Querying the Database

In the last chapter you learned how to load seed data and create records, now it's time to begin querying that data! In this chapter I'll introduce you to Active Record's array of extraordinarily powerful query features. By the conclusion of this chapter you'll know how to efficiently query for all records, retrieve a specific record according to primary key, select only desired columns, order, group and limit results, retrieve random records, and paginate results. You'll also learn how to integrate these queries with the application controller and view to create listing and detail pages. Always striving towards writing readable code, we'll also enhance the models using a fantastic feature known as a *scope*.

[7] http://rspec.info

[8] https://github.com/thoughtbot/factory_girl

Chapter 4. Introducing Associations

Building and navigating table relations is an standard part of the development process even when working on the most unambitious of projects, yet this task is often painful when working with many web frameworks. Not so with Rails. Thanks to a fantastic feature known as *Active Record Associations*, defining and traversing these associations is a fairly trivial matter. In this chapter I'll show you how to define, manage, and interact with the `belongs_to`, `has_one`, `has_many`, `has_many :through`, and the `has_and_belongs_to_many` associations.

Chapter 5. Mastering Web Forms

Your application's web forms will preferably interact with the models, meaning you'll require a solid grasp on Rails' form generation and processing capabilities in order to integrate with your models in the most efficient way possible. While creating simple forms is fairly straightforward, things can complicated fast when implementing more ambitious solutions such as forms involving multiple models. In this chapter I'll go into extensive detail regarding how you can integrate forms into your Rails applications, covering both Rails' native form building solutions as well as several approaches offered by popular gems.

Chapter 6. Debugging and Optimizing Your Application

While it's fun to brag about all of your project's gloriously cool features, they came at great expense of your time and brainpower. Much of the effort was likely grunt work, spent figuring out problems such as why a query was running particularly slow or the reason a complex association wasn't working as expected. This important but decidedly unglamourous part of programming is something I try to minimize on every occasion by relying on a number of tools and techniques that can automate much of the analysis and debugging process. In this chapter I'll introduce you to several of my favorite solutions for debugging and optimizing Rails applications.

Chapter 7. Integrating User Accounts with Devise

Offering users the opportunity to create an account opens up a whole new world of possibilities in terms of enhanced interactivity and the opportunity to create and view custom content. Yet there are quite a few moving parts associated with integrating even basic account features, including account registration, secure password storage, sign in, sign out and password recovery interfaces, and access restriction. Fortunately the fantastic Devise gem greatly reduces the amount of work otherwise required to implement account creation, authorization and management features, and in this chapter I'll introduce you to many of the wonderful features this gem has to offer.

Introducing the ArcadeNomad Theme Project

In my recent book, “Easy PHP Websites with the Zend Framework”, I based the material around the development of a real-world project known as GameNomad, a social networking application for console and PC gamers. This approach proved to be such a hit with readers that I thought it would be fun to implement a similar project for this book. This time however I’ve gone retro and created ArcadeNomad[9], an application which catalogs locations (bars, restaurants, laundromats, etc.) which house one or more so-called old school arcade games like the ones I spent so much time playing as a child in the 80’s. After all, who isn’t always up for a game of Space Invaders, Pac-man or Donkey Kong? Yet these games are increasingly difficult to find in public, and so hopefully ArcadeNomad will help fellow retro-gamers relive some great memories.

Of course, keep in mind this book is about Rails’ Active Record implementation and not ArcadeNomad, so while the book will base many of the examples upon the code found in the sample application, in some cases the example code will be simplified or modified for the sake of instruction. It’s just not possible to offer an exhaustive introduction to all facets of the ArcadeNomad codebase, however each example will stand on its own in terms of helping you to understand the topic at hand. If you purchased the book package that includes the ArcadeNomad source code then by all means open the appropriate files in your editor as we proceed through the book; otherwise if you purchased solely the book then you’re going to get along just fine. You can always return to http://easyactiverecord.com[10] at your convenience to purchase just the source code separately if you choose to do so.

> If you’ve purchased the book on Amazon, BN.com, or elsewhere and would like to additionally purchase the ArcadeNomad code, I’ve created a special discount code so you can save the same amount of money as somebody buying the book package on http://easyactiverecord.com. Go to http://easyactiverecord.com and use the Gumroad offer code `amazon` to buy the ArcadeNomad code for just $9.

All readers can interact with ArcadeNomad over at http://arcadenomad.com/[11]. This is a responsive application based on Bootstrap[12] built with mobile users in mind, however the application works just fine on a tablet or laptop too. Be sure to spend some time playing with the features in order to have a better idea of the sorts of data-related examples I’ll be presenting throughout this book. And by all means if you know of any locations with an Arcade game or two, be sure to add them!

About the Author

W. Jason Gilmore is a web developer, writer, and business consultant with more than 17 years of experience helping companies large and small build amazing software solutions. He is the author

[9] http://arcadenomad.com

[10] http://easyactiverecord.com

[11] http://arcadenomad.com/

[12] http://getbootstrap.com/

of seven books, including the bestselling "Beginning PHP and MySQL, Fourth Edition"[13] and "Easy PHP Websites with the Zend Framework, Second Edition"[14].

Over the years Jason has published more than 300 articles within popular publications such as Developer.com, PHPBuilder.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. He's recently led the successful development and deployment of a 10,000+ product e-commerce project, and is currently the lead developer on an e-commerce analytics project for a major international book publisher. Jason is cofounder of the wildly popular CodeMash Conference[15], the largest multi-day developer event in the Midwestern United States.

Jason loves talking to readers and invites you to e-mail him at wj@wjgilmore.com.

Errata and Suggestions

Nobody is perfect, particularly when it comes to writing about technology. I've surely made some mistakes in both code and grammar, and probably completely botched more than a few examples and explanations. If you found an error in the ArcadeNomad code base, or would like to report an error found in the book (grammatical, spelling, or instructional), please e-mail me at wj@wjgilmore.com.

[13] http://www.amazon.com/Beginning-PHP-MySQL-Professional-Development/dp/1430231149/

[14] http://www.amazon.com/Easy-PHP-Websites-Zend-Framework-ebook/dp/B004RVNL3G/

[15] http://www.codemash.org/

Chapter 1. Introducing Models, Schemas and Migrations

A well-designed web application will be *model-centric*, meaning the models that form the crux of the application (for instance, games and locations) will be heavily involved in the application's CRUD (create, retrieve, update, and delete) operations. Fortunately, the Rails framework excels at providing developers with the tools and features necessary to build and manage powerful models and their respective underlying database tables.

In this opening chapter I'll offer a wide ranging introduction to these tools and features, showing you how to build, populate, and test several basic versions of models used within the ArcadeNomad application. We'll discuss the various facets of model generation, schema management using a great feature known as *migrations*, why and how you might override various model defaults such as table naming conventions, how to configure the fantastic RSpec behavior-driven development tool and other utilities in order to effectively test your models, and finally how to create your first tests.

Creating the ArcadeNomad Project

Let's kick things off by creating the ArcadeNomad project, which will serve as the thematic basis for most of the examples found throughout this book. Keep in mind that I'm using Rails 4 for all examples found throughout this book, so you may occasionally encounter an output discrepancy if you're still using Rails 3. To create the project, use your operating system terminal to navigate to the desired location of the ArcadeNomad project directory and execute this command:

```
$ rails new dev_arcadenomad_com
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      ...
      Using sass (3.2.9)
      Using sass-rails (4.0.0)
      Using sdoc (0.3.20)
      Using sqlite3 (1.3.7)
      Using turbolinks (1.3.0)
      Using uglifier (2.1.2)
      Your bundle is complete!
      Use `bundle show [gemname]` to see where a bundled gem is installed.
```

I've omitted the majority of this command's output, which echoes various tasks related to the creation of project directories and files and the downloading and installation of project gems. Presuming you meet the reader requirements defined in the book's introduction then much of this will be familiar. However the Active Record-related features may be unfamiliar, so let's review a few related key files and directories created during this phase:

- The `app/models` directory houses the classes that define the entities you'll manipulate in your application. For instance, the ArcadeNomad application involves games and locations, therefore the `models` directory will soon house class files named `Game.rb` and `Location.rb`, respectively. At present however you'll find this directory to logically be empty of any class files, because we haven't yet created any models.
- The `db` directory contains various files used to manage your database schema and data, including the actual database location if you use the default SQLite database to manage your application data. At present this directory contains but a single file titled `seeds.rb` that serves as a central location for defining the application's initial data. I'll introduce the `seeds.rb` file in the next chapter.
- The `config/database.yml` file defines the configuration credentials used to connect to your project's development, test, and production database. You'll notice all three point to SQLite databases, because Rails' default supported database is indeed SQLite (http://www.sqlite.org/[16]). SQLite a perfectly acceptable database for many uses however chances are you're going to want to upgrade to MySQL (http://www.mysql.com/[17]) or PostgreSQL (http://www.postgresql.org/[18]). I'll show you how to configure MySQL and PostgreSQL in the following section.

Configuring the Database

Rails applications are configured by default to use the SQLite database. While SQLite is a perfectly capable database solution, (see http://www.sqlite.org/famous.html[19] for a list of well-known users), the majority of Rails developers prefer to use MySQL or PostgreSQL, and so in this section I'll show you how to configure both database gems.

Installing the MySQL Gem

To use MySQL, you'll need to update your project `Gemfile` to add MySQL support. Open your project's `Gemfile` (located in your project's root directory) and locate the following lines:

[16] http://www.sqlite.org/

[17] http://www.mysql.com/

[18] http://www.postgresql.org/

[19] http://www.sqlite.org/famous.html

```
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
```

Directly below these lines, add the following lines:

```
# Use mysql2 as the database for Active Record
gem 'mysql2'
```

Keep in mind you're not required to reference this gem within the Gemfile at precisely this location; I just prefer to group like-minded gems together for organizational purposes. Additionally, this will *only* install the gem, allowing your Rails application to talk to MySQL; it does not install the MySQL server. The steps required to install MySQL will vary according to your operating system; consult the MySQL documentation[20] for instructions.

After saving the changes to `Gemfile`, run `bundle install` from within your project's root directory to install the gem.

Installing the PostgreSQL Gem

Although all of the examples found throughout this book have been tested exclusively within MySQL, except for a very few exceptions (which I'll point out as applicable) I see no reason why they won't work equally well using a PostgreSQL backend. To use PostgreSQL you'll need to install the PostgreSQL gem. Open your project's `Gemfile` (located in your project's root directory) and find the lines:

```
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
```

Directly below these lines, add the following lines:

```
# Use PostgreSQL as the database for Active Record
gem 'pg'
```

Keep in mind you're not required to reference this gem within the Gemfile at precisely this location; I just prefer to group like-minded gems together for organizational purposes. Additionally, this will *only* result in installation of the gem, which allows your Rails application to talk to PostgreSQL; it does not install the database server. The steps required to install PostgreSQL will vary according to your operating system; consult the PostgreSQL documentation[21] for instructions.

After saving the changes to `Gemfile`, run `bundle install` from within your project's root directory to install the gem.

[20] http://dev.mysql.com/doc/

[21] http://www.postgresql.org/docs/

Configuring the Database Adapter

You'll need to supply the necessary credentials in order to connect to and interact with your project database. Keeping with Rails' DRY (Don't Repeat Yourself) principle, these credentials are stored in a single location and retrieved as needed. To make your database credentials available to the newly created ArcadeNomad application, open the file `config/database.yml`, which by default looks like this:

```
# SQLite version 3.x
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem 'sqlite3'
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

This file is broken into three distinct sections, including `development`, `test`, and `production`. These sections refer to the three most common phases of your application lifecycle. Because we're currently working in the development environment, you'll want to modify the following section:

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

SQLite database access permissions are managed by the underlying operating system (via file permission settings) rather than through a database-specific solution, meaning there is no need to supply a username or password. If you plan on using MySQL or PostgreSQL, you'll need to adjust this `development` setting a bit to accommodate the slightly more complex connection requirements.

Configuring Your Application for MySQL

When using MySQL you'll typically want to define a username, password, and host (the location from which the connecting user originates, typically `localhost`) in order to connect to the database. To do so, replace the `development` section with the following, modifying the `adapter` field to identify the desired database type (`mysql2`) and the `database` field to identify the name of your development database (you can call it anything you please however I prefer to use a convention that identifies the environment and domain, so in this case `dev_arcadenomad_com`). You'll also need to add `username`, `password`, and `host` fields to define the connecting user's username, password, and host, respectively:

```
development:
  adapter: mysql2
  database: dev_arcadenomad_com
  username: arcadenomad_dev
  password: supersecret
  host: localhost
```

Save the `database.yml` file and then create the `arcadenomad_dev` user if it doesn't already exist, granting all privileges for the database `dev_arcadenomad_com` in the process. You can do so by logging into the `mysql` client as the MySQL `root` user and execute the following command:

```
mysql>grant all privileges on dev_arcadenomad_com.* to arcadenomad_dev@localhost
     >identified by 'supersecret';
```

Although I prefer to do most MySQL administrative tasks using the terminal client, you can easily perform the same operation using phpMyAdmin or a similar application.

Once you've completed this step, move on to the section "Creating the Application Database".

Configuring Your Application for PostgreSQL

Like MySQL, in order to connect to PostgreSQL you'll need to identify the appropriate adapter. Modify the `development` section to look like this:

```
development:
  adapter: postgresql
  database: dev_arcadenomad_com
  pool: 5
  timeout: 5000
```

I'm taking a bit of a shortcut in this example because the PostgreSQL installer will by default create a password-less user having the same name as the user who was logged in at the time of installation. If a username is not supplied within the `database.yml` file, Rails will attempt to access PostgreSQL as the currently logged-in user. After you've updated the `development` section, save the file and proceed to the section "Creating the Application Database". If you're new to configuring PostgreSQL be sure to check out the PostgreSQL documentation[22].

Creating the Application Database

You can use Rake to create your database (presuming you haven't already done so using the `mysql` client or a similar application such as phpMyAdmin or MySQL Administrator) by opening a console, navigating to your project root directory, and executing the following command:

```
$ bundle exec rake db:create
```

Successful execution of this command does not produce any output, so you can confirm your Rails' application's ability to connect to the newly configured MySQL database by executing the `rake db:version` command, which is actually used to learn more about the state of your database schema (more about this later), and in doing so requires the ability to properly connect to your database using the `database.yml` parameters:

```
$ bundle exec rake db:version
Current version: 0
```

If you receive an ugly connection error regarding denied access, confirm your database credentials have been properly specified within the `database.yml` file. Otherwise you'll receive the message `Current version: 0`, which reveals the current schema version of your database. This should logically be `0` since we've yet to do anything with it (more about schema versions later in this chapter). Barring any problems there, confirm the MySQL account has been properly configured and that the MySQL database server is currently running.

[22] http://www.postgresql.org/docs/

Saving Keystrokes with Binstubs

You can decrease the amount of typing required to run `rake` and `rspec` commands by eliminating the need to prefix them with `bundle exec`. Save some keystrokes by running the following two commands:

```
$ bundle binstubs rspec-core
$ bundle binstubs rake
```

This will allow you to execute Rake tasks and RSpec tests by running `rake` and `rspec`, rather than `bundle exec rake` and `bundle exec rspec`, respectively. Because I'm lazy I'll presume you've done this and moving forward will omit `bundle exec` when executing Rake and RSpec examples, so if you for some reason opt to not perform this step then be sure to prefix your `rspec` commands with `bundle exec`.

Installing the RSpec, FactoryGirl and Database Cleaner Gems

Next we'll add the RSpec, FactoryGirl and Database Cleaner gems, all of which are indispensable for testing your Rails models. For the moment we'll just focus on installing these gems, and will return to the topic at the end of this chapter. Scroll down below the `assets` group and add the following section:

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0.0'
  gem 'factory_girl_rails', '~> 4.4.0'
  gem 'database_cleaner', '~> 1.3.0'
end
```

By placing the reference within a `group` block, you'll limit availability of the gem to specifically the `development` and `test` environments, omitting it from the `production` environment since you won't be executing tests when running the application in production mode. Save the file and return to the command line, executing the following command to install these gems:

```
$ bundle install
```

Unless you received rather explicit error messages stating otherwise, chances are high the gems were correctly installed. To soothe any paranoia, you can confirm a gem has been successfully installed using the `bundle show` command as follows:

```
$ bundle show rspec-rails
```

After installation has completed, you'll need to generate a few directories which will house your test scripts:

```
$ rails generate rspec:install
  create  .rspec
  create  spec
  create  spec/spec_helper.rb
  create  spec/rails_helper.rb
```

Congratulations, you're ready to begin testing your models! We'll return to this topic at the end of the chapter.

Introducing Rails Models

With the ArcadeNomad project created and MySQL database configured, it's time to start working on the models, the most fundamental of which is that used to manage the locations housing the arcade games. This model will evolve substantially over the course of the next few chapters, however as you'll soon see we'll be able to build some pretty cool features even when getting acquainted with Rails' fundamental model management capabilities.

Creating a Model

Let's start by creating what is perhaps the most fundamental model of the ArcadeNomad project, the `Location` model. You'll generate models using the `generate` command. In the first of many hands-on exercises found throughout the book, let's generate the `Location` model. Execute the following command from the root directory of your newly created ArcadeNomad project:

```
$ rails generate model Location
  invoke  active_record
  create    db/migrate/20140724130112_create_locations.rb
  create    app/models/location.rb
  invoke    rspec
  create      spec/models/location_spec.rb
  invoke      factory_girl
  create       spec/factories/locations.rb
```

Congratulations, you've just created your first model! Incidentally, you can save a few keystrokes by using the `generate` shortcut, `g`:

```
$ rails g model Location
```

In either case, this command creates four important files, including the model (`app/models/location.rb`), and the migration (`db/migrate/20140724130112_create_locations.rb`). We will soon use the migration file to generate the model's associated database table schema. The third file (`spec/models/location_spec.rb`) is used to test the `Location` model's behavior, and the fourth (`spec/factories/locations.rb`) is used to conveniently generate model objects for use within the tests. We'll return to all of these files throughout the chapter.

Mind the Model Naming Conventions

Rails model names should be singular and camel case. Therefore, suitable model names would include `Game`, `Location`, `User`, and `SupportTicket`. Unsuitable names include `Games`, `Users` and `Gamescomments`.

Creating the Model's Corresponding Database Table

Models aren't particularly useful without a corresponding database table schema. Recall that a file known as a *migration* was created when the model was generated. Migrations offer an incredibly convenient means for evolving a database over time using a Ruby DSL (Domain Specific Language) and a variety of Rake commands that make it incredibly easy to transition the database schema from one version to the next. Using this DSL you can create and drop tables, manage table columns, and add column indexes, among other tasks. Further, because each migration is stored in a text file, you can manage them within your project repository. Let's use one of these Rake commands to migrate the first set of changes (creating the `Location` model's corresponding `locations` table):

```
$ rake db:migrate
==  CreateLocations: migrating ======================================
-- create_table(:locations)
   -> 0.0116s
==  CreateLocation: migrated (0.0117s) =================================
```

Presuming you're seeing output similar to that shown above, rest assured the `locations` table has indeed been successfully created. However, because this is possibly your first time using Active Record's migrations feature, login to your MySQL database using phpMyAdmin or your MySQL client and confirm the table has indeed been created. I'm a command-line kind of guy, and so for demonstration purposes will use the MySQL client, however if you're not using MySQL you can use whatever database-specific solution you prefer to achieve the same result:

```
$ mysql -u arcadenomad_dev -p dev_arcadenomad_com
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 62
Server version: 5.5.9-log Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> show tables;
+-------------------------------+
| Tables_in_dev_arcadenomad_com |
+-------------------------------+
| locations                     |
| schema_migrations             |
+-------------------------------+
```

It may come as a surprise to see the database consists of not one but two tables: `locations` and `schema_migrations`. The `schema_migrations` table is responsible for keeping track of which migrations have been applied. Review the table contents to see what is inside:

```
mysql> select * from schema_migrations;
+----------------+
| version        |
+----------------+
| 20140724130112 |
+----------------+
```

Returning to the output generated by the `rails g model Location` command, you'll see the timestamp found in the `schema_migrations` table matches that prefixing the migration file which was generated at that time (`db/migrate/20140724130112_create_locations.rb`). As you perform future migrations, the timestamps attached to those files will be appended to this table, with the largest timestamp logically inferring which migration was most recently executed. This is important because it lets Rails know which migration should be reverted should you decide to undo the most recent changes. Give it a try by returning to your project's root directory and run the `rollback` command:

```
$ rake db:rollback
==  CreateLocations: reverting =================================
-- drop_table("locations")
   -> 0.0333s
==  CreateLocations: reverted (0.0333s) ============================
```

After rolling back your latest change, return to the database and view the table listing anew:

```
mysql> show tables;
+---------------------------------+
| Tables_in_dev_arcadenomad_com |
+---------------------------------+
| schema_migrations             |
+---------------------------------+
```

Sure enough, the `locations` table has been deleted. Of course, we want the `locations` table after all, so return to the console and run that latest migration one more time:

```
$ rake db:migrate
==  CreateLocations: migrating ==================================
-- create_table(:locations)
   -> 0.0837s
==  CreateLocations: migrated (0.0838s) ============================
```

I'm a big fan of migrations because they offer a rigorous way to manage the evolution of a database over time. Logically, the migration files are stored in version control alongside all other source files, which among other benefits provides other members of your team with an easy way to synchronize the latest database changes with their own local development environments. We'll return to the matter of migrations in the later section, "Introducing Migrations", where I'll offer an in-depth introduction to migrations syntax and commands.

With this brief but important tangent complete, let's finally return to the topic at hand: the `locations` table. Return one last time to your database and review the `locations` table structure:

```
mysql> describe locations;
+------------+----------+------+-----+---------+----------------+
| Field      | Type     | Null | Key | Default | Extra          |
+------------+----------+------+-----+---------+----------------+
| id         | int(11)  | NO   | PRI | NULL    | auto_increment |
| created_at | datetime | NO   |     | NULL    |                |
| updated_at | datetime | NO   |     | NULL    |                |
+------------+----------+------+-----+---------+----------------+
```

The `locations` table currently consists of just three fields, `id`, `created_at`, and `updated_at`. These fields are added by default to every table generated using migrations, unless you so choose to override the defaults (see the later section, "Overriding Migration Defaults"). The `id` field is an auto-incrementing integer that serves as the table's primary key. The `created_at` and `updated_at` fields serve as timestamps (although they are managed using the `datetime` data type), identifying the precise date and time in which the row was created and last modified, respectively. You don't actually have to manually manipulate any of these three fields when interacting with them via Active Record, because Active Record will manage them for you!

Of course, we'll want to add all sorts of other useful fields to the `locations` table, such as the location title, description, street address, and phone number. We'll add these fields throughout the remainder of this chapter as new concepts are introduced.

Rails Models are Plain Old Ruby Objects

Believe it or not, a Rails model is simply a Ruby class that subclasses the `ActiveRecord::Base` class. In subclassing `ActiveRecord::Base`, the class is endowed with a variety of features useful for building powerful Rails models, but it otherwise behaves exactly as you would expect from a standard Ruby class, meaning you can add both class and instance methods and properties, among other things. Open the file `app/models/location.rb` and Rails 4 users will see the following class skeleton:

```
class Location < ActiveRecord::Base
end
```

Rails 3 users will see an almost identical skeleton, save for the additional commented out `attr_accessible` macro:

```
class Location < ActiveRecord::Base
  # attr_accessible :title, :body
end
```

The `attr_accessible` macro is a Rails 3 feature that explicitly identifies (or "whitelists") any model attributes that can be updated via any of Rails' various mass assignment methods. These mass assignment methods can make record manipulation quite convenient however must be used with caution as they open up the possibility of an attacker maliciously manipulating the form with the intent of modifying a potentially sensitive attribute. Rails 4 departs from this approach in a fairly radical fashion, requiring developers to instead identify those attributes which are approved for mass assignment from within the appropriate *controller* rather than model. In the next chapter you'll find a section titled "Introducing Strong Parameters" where I'll explain the pros and cons of the `attr_accessible` macro and why the Rails 4 development team opted for a different approach.

Because a Rails model is just a Ruby class, you can instantiate it using the `new` method. To demonstrate this, navigate to your project's root directory and login to the Rails console:

```
$ rails console --sandbox
Loading development environment in sandbox (Rails 4.1.1)
Any modifications you make will be rolled back on exit
```

Notice I've invoked the Rails console in sandbox mode. This is a very useful feature because it gives you the opportunity to experiment with your application's models without worrying about any lasting effects. After exiting the console (by executing the `exit` command), any changes you had made during the console session will be negated. Try creating a new instance of the `Location` model by invoking the `new` method:

```
>> Location.new
 => #<Location id: nil, created_at: nil, updated_at: nil>
```

The Rails Console

I find the Rails Console to be an indispensable tool, and leave a session open almost constantly throughout the day. It is supremely useful for easily and quickly experimenting with models, queries, and debugging various other data structures such as arrays and hashes. You might have noticed the above Rails console prompt is a tad more stark then yours, presuming you haven't already changed the default prompt. The default prompt includes the Ruby version and command number, neither of which I find to be useful. If you would like a simplified console prompt create a file named `.irbrc` and place it in your home directory. Inside it, add the following statement:

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
```

After saving the file exit and re-enter the console and you'll see the simplified prompt!

When invoking `Location.new`, the console returns the object contents, which not surprisingly consists of the three properties defined in the associated `locations` schema, all of which are set to `nil`. Of course, if you want to actually interact with an object you'll first need to create one:

```
>> location = Location.new
 => #<Location id: nil, created_at: nil, updated_at: nil>
```

Remember, you don't actually manipulate the `id`, `created_at`, and `updated_at` properties, because Active Record will handle them for you. Let's go ahead and persist this object within the `locations` table:

```
>> location.save
  (0.4ms)  SAVEPOINT active_record_1
  SQL (21.7ms)  INSERT INTO `locations` (`created_at`, `updated_at`)
   VALUES ('2014-07-25 02:17:05', '2014-07-25 02:17:05')
  (0.2ms)  RELEASE SAVEPOINT active_record_1
=> true
```

Once saved, the persisted property values are immediately made available to the object, as demonstrated here:

```
>> location
=> #<Location id: 1, created_at: "2014-07-25 02:17:05",
     updated_at: "2014-07-25 02:17:05">
```

Of course, you can also reference the object properties individually:

```
>> location.created_at
=> Thu, 25 Jul 2014 02:17:05 UTC +00:00
```

You'll likely want to return at some later point in time to retrieve the recently added `Location` object. One of the most basic ways to do so is by identifying the object by its primary key (the `id` column) using the `find` method (the `find` method is formally introduced in Chapter 3):

```
>> location = Location.find(1)
Location Load (7.7ms)  SELECT `locations`.* FROM `locations`
  WHERE `locations`.`id` = 1 LIMIT 1
=> #<Location id: 1, created_at: "2014-07-25 02:17:05",
    updated_at: "2014-07-25 02:17:05">
```

Because the model is just a Ruby class, you are free to enhance it in ways which render the model more powerful and convenient to use. In later chapters we'll return to this capability, adding a variety of helper methods and other features to the ArcadeNomad models. For the moment let's add an example instance method to get a feel for the process. Specifically, we'll override the `to_s` method, providing an easy way to dump the object's contents in a human-readable format. Add the following `to_s` method to the `Location` model. In the meantime, do not exit the Rails console! I'm going to show you a useful trick involving making ongoing updates to a class and the console. Begin by adding the `to_s` method:

```
class Location < ActiveRecord::Base

  def to_s
    "#{id} - Created: #{created_at} - Updated: #{updated_at}"
  end

end
```

Now, return to your Rails console, and attempt to reload the object and call the newly created `to_s` method:

```
>> location = Location.find(1)
Location Load (0.3ms)  SELECT `locations`.* FROM `locations`
  WHERE `locations`.`id` = 1 LIMIT 1
=> #<Location id: 1, created_at: "2014-07-25 02:17:05",
   updated_at: "2014-07-25 02:17:05">
>> location.to_s
NoMethodError: undefined method `to_s' for #<Location:0x007ffb8f2523c0>
```

Why did this happen? You presumably saved your changes, meaning the `to_s` method is indeed part of the model. It's because the Rails console needs to be made aware any such changes have occurred, done by executing the `reload!` command:

```
>> reload!
Reloading...
 => true
>> location = Location.find(1)
 Location Load (1.0ms)  SELECT `locations`.* FROM `locations`
  WHERE `locations`.`id` = 1 LIMIT 1
=> #<Location id: 1, created_at: "2014-07-25 02:17:05",
   updated_at: "2014-07-25 02:17:05">
>> location.to_s
=> "1 - Created: 2014-07-25 02:17:05 UTC - Updated: 2014-07-25 02:17:05 UTC"
```

Fantastic! You're now aware of the ability to expand model capabilities with instance methods. But your capabilities of course don't stop here, because you're free to add class methods, as well as instance and class attributes. If you're new to object-oriented Ruby or require a refresher, I suggest checking out the Ruby Programming Wikibook[23]. In any case, you'll become much more familiar with these capabilities as we continue expanding the various ArcadeNomad model capabilities throughout this book.

[23] http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Classes

Defining Accessors and Mutators

Rails will conveniently add default *setters* and *getters* (also known as *mutators* and *accessors*, respectively, although I'll use the former terminology throughout the book) to your models' data attributes, meaning you can easily assign and retrieve values to the associated columns using dot notation syntax. For instance, the following example will assign and retrieve the name of an instance of the `Game` model:

```
>> game = Game.new
>> game.name = 'Space Invaders'
>> puts game.name
Space Invaders
>>
```

There will be occasions where an attribute's value is not yet in a state where it's ready for assignment, yet isn't of a severity that a validation error is warranted (validations are introduced in the next chapter). Consider the typical arcade location found in ArcadeNomad. When identifying a new location, the user can optionally include a telephone number. Of course, a phone number can take many valid forms, `(614) 555-1212`, `614.555.1212` and `614 555-1212` among them. When a user decides to add a new location to the database, logically we want to empower the user to do so in the most convenient fashion possible. Of course, one could construct a series of form fields that requires the user to input the area code, prefix and line number separately, however particularly when using a mobile device it would be more convenient to allow the user to enter the number in any manner they please provided what is enters consists of ten total digits. At the same time, we want to store *only* the digits in the `Location` model's `telephone` field. Therefore if the user enters `(614) 555-1212` we only want `6145551212`. You can satisfy both requirements by defining a custom setter to strip out any unwanted characters:

```
def telephone=(value)
  write_attribute(:telephone, value.gsub(/[^0-9]/i, ''))
end
```

The setter is just a standard Ruby instance method, but it must be assigned the same name as the model data attribute whose setter you'd like to override. The `write_attribute` method is used to actually set the value. In this example Ruby's `gsub`[24] method is used to strip out anything that's not an integer. Once set you can use the console to test the custom setter:

[24] http://apidock.com/ruby/String/gsub

```
>> reload!
>> location = Location.new
...
>> location.telephone = '(614) 555-1212'
"(614) 555-1212"
>> location.telephone
>> 6145551212
```

Custom getters are defined in the same manner as setters, although you won't be passing in a parameter since the idea is to retrieve a value rather than set one. However, determining when it is appropriate to use a custom getter is a much more nebulous process, because in many cases more convenient (and proper) alternative solutions exist. In any case let's take a look at how one is constructed. You'll define a custom getter by creating a method and setting its name identically to the name of the attribute whose default getter you're trying to override, using the `read_attribute` method to retrieve the attribute's value. Consider a scenario where a future version of ArcadeNomad offered user registration and profiles, and these user profiles encouraged users to upload an avatar image. If uploaded, the avatar image path and name would be stored in an attribute named `avatar`. Otherwise, a default image would be set using a path defined in `default_avatar`. A custom getter would make this conditional process the default when retrieving the `avatar` attribute:

```
def avatar
  read_attribute('avatar') || default_avatar
end
```

I stated the motivations for using a custom getter can quickly become murky because it is easy to misuse them. For instance, you should *not* use custom getters to determine how a value should be formatted for the user. Returning to the `Location` model's `telephone` attribute, clearly you're going to want to present a location's phone number in a format more user-friendly than how the numbers are stored (e.g. `6145551212`). However, you should use a *view helper* for such purposes rather than a custom getter. See the Rails documentation for more information about view helpers.

Introducing Virtual Attributes

Sometimes you'll want to create a different representation of one or more fields without incurring the administrative overhead of managing another model attribute. You can create a *virtual attribute* by combining multiple attributes and returning them in a combined format. For instance, each ArcadeNomad location is associated with a street address, city, state and zip code. What if you wanted the convenience of simply referencing an `address` attribute in order to retrieve a string that looks like `254 South Fourth Street, Columbus, Ohio 43215`? You can create a method that does this for you:

```
class Location < ActiveRecord::Base

  ...

  def address
    street + ' ' + city + ', ' + state + ' ' + zip
  end

end
```

Note for the sake of illustration I'm assuming the `address`, `city`, `state` and `zip` attributes are all strings. In the real world it's a bit more complicated than this because the state would typically be normalized within its own table. We'll tackle such complications soon enough, so for the moment just roll with it. Once define you can reference the address like this:

```
>> reload!
>> l = Location.find(8)
>> l.address
  => "254 South Fourth Street, Columbus, Ohio"
```

Introducing Migrations

The previous section introduced you to Active Record migrations, explaining how a model's corresponding schema migration is automatically generated, and how migrations can be both deployed and rolled back using your project's companion Rake commands. Although important, what you've learned about migrations thus far is but a taste of what you can do with this fantastic feature. In this section I'll show you how to use migrations to easily evolve your database over the project's lifetime.

Anatomy of a Migration File

A migration file consists of a series of table alteration commands using a custom DSL (domain-specific language) which eliminates the need for developers to grapple with SQL syntax. For instance, the following migration file was created when you generated the `Location` model earlier in this chapter:

```
class CreateLocations < ActiveRecord::Migration
  def change
    create_table :locations do |t|

      t.timestamps
    end
  end
end
```

As you can see, the migration file is nothing more than a Ruby class that inherits from the `ActiveRecord::Migration` class. Like any other typical migration, it consists of a single method named `change` that defines the schema alteration statements. In this case these statements are in turn encapsulated within a `create_table` definition that identifies the name of the table to be created (`locations`) as well as the columns. This latter bit of instruction (`t.timestamps`) is admittedly rather confusing due to a bit of artistic freedom exercised by Rails in that the `t.timestamps` declaration actually creates *two* table columns, including `created_at` and `updated_at`, both of which are defined as `datetime` data types and which are autonomously managed by Rails. This means when a new record is created, the `created_at` column will automatically be updated to reflect the date and time in which the record was created. When the record is later updated, the `updated_at` column will be updated to reflect the date and time in which the update occurred.

The `change` method is special in that should you choose to subsequently roll back any changes made by the migration, in most instances Rails knows how to reverse the changes. For instance if the `create_table` definition is used within the `change` method, then Rails knows to drop the table should the migration be rolled back. The `change` method isn't infallible however, as it is incapable of reversing more complicated changes than those which going beyond standard schema alterations. While we won't be delving into anything not supported by `change` in this book, should you have more esoteric needs be sure to consult the Rails manual regarding the `up` and `down` methods and the `reversible` feature.

You'll note the blank line between the `create_table :locations do |t|` and `t.timestamps` statements. This is not a typo, but is rather the place where you would optionally insert other column definitions. For instance to add a `name` column to the `locations` table you'll update this file to look like this:

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

In the following sections I'll introduce you to the meaning of `t.string` and offer more convenient ways to associate columns with your database tables.

Useful Migration Commands

You've already learned how to migrate (`rake db:migrate`) and rollback changes (`rake db:rollback`), however there are several other useful Rake commands at your disposal. I won't review every available command, however recommend you take a moment to peruse all available commands by executing `rake -T db`.

You can check the status of your migrations using the `db:migrate:status` command:

```
$ rake db:migrate:status

database: dev_arcadenomad_com

 Status   Migration ID    Migration Name
--------------------------------------------------
   up     20140724130112  Create locations
   up     20140823043933  Create games
   up     20140825024130  Add address columns to location
   up     20140825024435  Create states
   down  20140827014448  Create categories
```

This example output representing some future point in the ArcadeNomad development process indicates that the `Create games`, `Create locations`, `Add address columns to location`, and `Create states` migrations have been executed (denoted by the `up` status), while the `Create categories` migration has yet to be executed (denoted by the `down` status).

Rolling Back Your Changes

On of the beautiful aspects of migrations is the ability to easily undo your changes. For instance, suppose you've just executed the migration for a new model schema, only to immediately realize you forgot to include an attribute. As you'll soon learn you can add a new column using a separate migration, however in such cases I prefer to avoid cluttering up my migration list with unnecessary additional migrations and instead roll back (undo) the migration, add the desired attribute, and run the migration anew. To roll back the most recently executed migration you'll execute the `db:rollback` command:

```
$ rake db:rollback
```

If you've just executed several migrations only to realize you made a mistake in an earlier migration, you can roll back several migrations using the `STEP` parameter. For instance suppose you want to roll back the last two migrations:

```
$ rake db:rollback STEP=2
```

After correcting the earlier migration, merely running `db:migrate` will result in all migrations identified as being `down` being run anew.

Running Only a Specific Migration

Particularly in the early stages of a project you might rapidly create several models in succession, yet not be quite ready to immediately generate all of their respective schemas. Yet running `rake db:migrate` will run all outstanding migrations. You can run a specific migration by identifying its version number like so:

```
$ rake db:migrate:up VERSION=20140921124955
```

Datatypes, Attributes and Default Values

Active Record supports all of the data types you've grown accustomed to when working with a database such as MySQL. So far you've encountered `string`, `datetime` and `timestamp`, however as the following table indicates, there are plenty of other data types are at your disposal.

Supported Data Types

Datatype	MySQL Data Type
binary	blob
Boolean	tinyint(1)
date	date
datetime	datetime
decimal	decimal
float	float
integer	int(11)
string	varchar(255)
text	text
time	time
timestamp	timestamp

Of course, data types can only go so far in terms of defining a column; you'll also often want to further constrain the column by defining whether the column is nullable, specifying default values, limiting the column length, and setting a decimal column's precision and scale. You'll use one or several column options to do so, the most popular of which are defined in the following table.

Supported Column Options

Option	Definition
default	Define a column's default value
limit	Defines a column's maximum length; characters for `string` and `text`, bytes for `binary` and `integer`
null	Determines whether a column can be set to NULL
precision	Defines the precision for `decimal`
scale	Defines the scale for `decimal`

Understanding how these options are used is best explained using several examples. Presume you want to create a column to represent a user's current age. While it would be perfectly acceptable to use the `integer` data type, `integer` sports a pretty large range, -2147483648 to 2147483647 to be exact, requiring 4 bytes to do so. You can dramatically reduce the supported range by instead using a `tinyint` (supporting a range of -128 to 127) by setting the `integer` limit to `1`:

```
t.integer :quantity, :limit => 1
```

The `limit` option is equally useful for constraining the size of strings. By default Rails will set the maximum size of a string to 255 characters, but what if you intend to use a `string` for managing a product SKU which are alphanumeric and never surpass a size of 8 characters? You can set the `limit` accordingly:

```
t.string :sku, :limit => 8
```

Many experienced Rails developers aren't aware that it's possible to define attributes on the command-line, perhaps because the syntax isn't nearly as obvious. For instance you can define `integer` and `string` limits by defining their respective limits within curly brackets:

```
$ rails g model Product quantity:integer{1} sku:string{8}
```

What about monetary values, such as 17.99? You should typically use a `decimal` with a `precision` of `8` and a `scale` of `2`:

```
t.decimal :salary, :precision => 8, :scale => 2
```

It can be easy to forget what `precision` and `scale` define, but it's actually quite straightforward after repeating it a few times: The `precision` defines the *total* number of digits, whereas the `scale` defines the number of digits residing to the right of the decimal point. Therefore a decimal with a precision of `8` and scale of `2` can represent values of a size up to `999,999.99`. A decimal with a precision of `5` and scale of `3` can represent values of a size up to `99.999`.

You can specify a `decimal` column's precision and scale within the generator like so:

```
$ rails g model Game price:decimal{8.2}
```

Let's consider one last example. Suppose you used a `Boolean` for managing a column intended to serve as a true/false flag for the row. For instance when an ArcadeNomad user adds a new arcade to the database, a Boolean column called `confirmed` is set to `false` so I can easily filter the arcades on this flag in order to ensure the submission doesn't contain any spam. Therefore to play it safe we should always presume new arcade entries are unconfirmed (false) until an administrator otherwise expressly confirms the submission. We can rest assured the `confirmed` column will be set to `false` by defining it like so:

```
t.boolean :confirmed, :default => false
```

It is not possible to set a default Boolean value using the generator, meaning you'll need to open the migration file and specify the default manually.

Streamlining the Model Creation Process

You'll presumably have a pretty good idea of what schema attributes will make up the initial version of a model, so why not identify these attributes at the same time you generate the model? You can take a time-saving shortcut by passing along the attributes when generating the model, as demonstrated here. In the following example we'll generate the `Game` model which will house information about the various games:

```
$ rails g model Game name:string description:text
  invoke  active_record
  create    db/migrate/20140802135201_create_games.rb
  create    app/models/game.rb
  invoke    rspec
  create      spec/models/game_spec.rb
  invoke      factory_girl
  create        spec/factories/games.rb
```

After generating the model, open the migration file and you'll see the `name` and `description` columns have already been included, negating the need to manually add them:

```
class CreateGames < ActiveRecord::Migration
  def change
    create_table :games do |t|
      t.string :name
      t.text :description
      t.timestamps
    end
  end
end
```

Next, create the `games` table by migrating the schema:

```
$ rake db:migrate
```

As before, feel free to log into your database and have a look at the newly created `games` schema to confirm the desired attributes have been created as expected.

Modifying Existing Schemas

Logically you'll want to continue evolving a model schema over the course of a project. To do so you'll generate a standalone migration which will contain commands responsible for adding, changing and deleting columns, adding indexes, and performing other tasks. You'll generate standalone migrations in the same way other aspects of a Rails application are generated, via the `rails generate` command:

```
$ rails generate migration yourMigrationNameGoesHere
```

In this section I'll guide you through various aspects of migration-based schema modification, many of which you'll come to rely upon repeatedly for your projects. I'll focus on the migration syntax you're most commonly going to require when managing your schemas (numerous other migrations features will be introduced in later chapters), so please don't consider this overview to be exhaustive. Be sure to consult the appropriate Rails documentation[25].

Adding a Column

You'll want to add new columns to a schema as it becomes necessary to manage additional bits of data associated with your model. For instance, we'll logically want to attach a name and description to each location, so let's add a `name` and `description` attribute to the `Location` model. You can save some typing by passing along the desired schema column names and data types at the same time the model is generated. In the following example we'll add a `name` and `description` attribute to the `locations` schema:

```
$ rails g migration AddNameAndDescriptionToLocations name:string
description:string
```

Rails is intelligent enough to understand you would like to *add* a column to the *locations* table thanks to the migration title `AddNameAndDescriptionToLocations`. In doing so, it generates the following migration file:

```
class AddDescriptionToLocations < ActiveRecord::Migration
  def change
    add_column :locations, :name, :string
    add_column :locations, :description, :string
  end
end
```

After saving the file, run the migration per usual to add the columns:

```
$ rake db:migrate
```

Renaming a Column

You'll occasionally add a new column to a schema only to later conclude a little more thought should have been put into the column name. For instance, suppose you added the columns `lat` and `lng` to the `locations` schema (for managing a location's latitudinal and longitudinal coordinates), later deciding you should improve their readability and therefore rename them as `latitude` and `longitude`, respectively. Per usual you'll start by generating a new migration:

[25] http://guides.rubyonrails.org/migrations.html

```
$ rails g migration renameLatAndLngAsLatitudeAndLongitude
 invoke  active_record
 create db/migrate/20140926021404_rename_lat_and_lng_as_latitude_and_longitude.rb
```

Next open up the migration file and use the `rename_column` command to rename the columns:

```
class RenameLatAndLngAsLatitudeAndLongitude < ActiveRecord::Migration
  def change
    rename_column :locations, :lat, :latitude
    rename_column :locations, :lng, :longitude
  end
end
```

Removing a Column

You'll occasionally add a column to a schema only to later conclude the data it's intended to contain isn't useful, or you decide to normalize the data within a separate schema. To remove a column you'll use the `remove_column` command:

```
remove_column :games, :some_column_i_dont_need
```

Dropping a Table

Suppose after one too many glasses of Mountain Dew one Friday evening you decide to add a compendium of video game character bios to the site, only to realize the next morning how much additional work this would require. You can delete the table using the `drop_table` statement:

```
class DropTableVideoGameCharacters < ActiveRecord::Migration
  def change
    drop_table :video_game_characters
  end
end
```

Keep in mind that dropping a table does not result in deletion of the companion model. You'll need to manually remove the model and any other related references.

Beware the schema.rb File

An auto-generated file named `schema.rb` resides in your project's `db` directory. If you open it up, you'll see it contains all of the schema creation commands you defined within the various project migration files. For instance, after generating a `locations` table the file looks like this:

```
# encoding: UTF-8
# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record to
# incrementally modify your database, and then regenerate this schema definition.
#
# Note that this schema.rb definition is the authoritative source for your
# database schema. If you need to create the application database on another
# system, you should be using db:schema:load, not running all the migrations
# from scratch. The latter is a flawed and unsustainable approach (the more
# migrations you'll amass, the slower it'll run and the greater likelihood
# for issues).
#
# It's strongly recommended that you check this file into your version control
# system.

ActiveRecord::Schema.define(version: 20140724130112) do

  create_table "locations", force: true do |t|
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

As the warning in the file comments makes clear, this file is *the authoritative source* for your database schema. Should you enlist the help of a fellow developer, it is likely that developer will want to run a local instance of the Rails application within his own development environment, and therefore will need to generate the project's database schema. To do so, the developer is highly encouraged to execute the following command to do so:

```
$ rake db:schema:load
```

In doing so, the database creation statements found `schema.rb` file will be used to create the tables. Further, this file is used to create the schema used for the test environment database (more about this later). Given these two important applications, do not delete `schema.rb` on the mistaken conclusion it's just a backup copy of your various migrations! The file actually serves a much more important purpose and therefore not only should you take care to not delete it, but you should also be sure to add it to your project's source repository.

Overriding Model and Table Defaults

Although you are strongly encouraged to abide by the Rails conventions regarding model and table names and structures, there are occasionally valid reasons for deviation. In this section I'll highlight two of the most common reasons for overriding these defaults.

Overriding the Table Name

Rails models are by default singular and camel case. For instance, valid model names include `User`, `Location`, and `GamePublisher`. The corresponding table name is always plural, lowercase, and use underscores as word separators. For instance, the table names corresponding to the aforementioned model examples are `users`, `locations`, and `game_publishers`, respectively. But what if you wanted to override a model's associated table name, for instance using the table name `configurations` in conjunction with the model `ConfigurationSetting`. You can override the table by setting the desired table name using the `table_name` method:

```
class ConfigurationSetting < ActiveRecord::Base

  self.table_name 'configurations'

end
```

Presumably in such a case you want to do this because the table already exists (although you could preferably opt to stick with conventions and rename the table). If you want to generate a model without the corresponding migration, you can set the `--migration` option to false:

```
$ rails g model ConfigurationSetting --migration=false
```

Overriding the Primary Key

Suppose you have no need for the typical auto-incrementing integer-based primary key, and desire to instead use a GUID (a practice which many find impractical but who am I to judge, particularly since none other than programming guru Jeff Atwood himself advocates the use of GUIDs as primary keys[26]). To override the default use of the `id` primary key column, you'll modify the `create_table` block within the migration file, disabling the `id` column and identifying the primary key column using the `primary_key` key:

```
create_table :games, :id => false, :primary_key => :guid do |t|
  t.string :guid, :null => false
  ...
end
```

Testing Your Models

Is it possible to save a game with a blank title? Is the user registration form being properly displayed? Surely it isn't possible for a location to not be associated with any games, right? Does the top twenty

[26] http://www.codinghorror.com/blog/2007/03/primary-keys-ids-versus-guids.html

most popular locations widget indeed display exactly twenty locations? These are just a few of the sorts of concerns developers are constantly facing, and for many developers the only way to satisfy these concerns is by constantly surfing the site and manually testing the various pages and features. This is a recipe for madness; not only is it time consuming but the entire process is guaranteed to be rife with errors and frustration.

Fortunately, Rails developers have several simple automated testing solutions at their disposal, which work together to provide peace of mind when it comes to answering these sorts of questions. One of the most popular solutions is called RSpec[27], and in this section I'll show you how to use RSpec and another great gem called FactoryGirl[28]. This being a book focusing on Active Record, logically much of the RSpec-related discussion will focus on testing Rails models, however I'll occasionally stray into other RSpec capabilities when practical, hopefully providing you with a well-rounded understanding of this powerful testing framework's capabilities.

Earlier in this chapter you installed RSpec and FactoryGirl. If you happened to skip this step and would like to follow along with the examples, consider circling back and installing these gems now. In any case, I'd imagine you'll be able to gain a pretty solid understanding of these gems' fundamental operation by simply reading along. Also, keep in mind this section merely serves as a friendly introduction to the topic, covering just enough material to help you get started writing simple tests; in subsequent chapters we'll build upon what's discussed here, taking advantage of more complex RSpec and FactoryGirl features.

Logically you might be wondering how the application models could even be tested at this point, given we've yet to even discuss how to save data or otherwise interact with the models. Indeed, there is little of a practical nature to discuss at this point, although there's plenty to review regarding readying your Rails application for subsequent testing.

Preparing the Test Database

Earlier in this chapter I introduced the `config/database.yml` file. To recap, this file defines the different connection parameters used to connect to your project's development, test, and production databases, in addition to any others you care to define. While introducing this section we reconfigured the default development database configuration to use MySQL or PostgreSQL, but left the test database configuration alone. By default the test database uses SQLite[29], a fast and lightweight database solution that in many cases can serve the role of housing the test database exceedingly well.

For reasons of convenience and the opportunity to introduce you to SQLite I'll just leave the current test database settings in place, however when working on any real-world project keep in mind you've chosen a framework such as Rails precisely because you want to avoid ugly surprises and other inconveniences borne out of loosely defined assumptions, so why risk some subtle difference between SQLite and your chosen database (if not SQLite) resulting in unnecessary hassle? Take a

[27] http://rspec.info/

[28] https://github.com/thoughtbot/factory_girl

[29] http://www.sqlite.org/

few extra minutes to update the test database environment to use the same database server you'll be using in development and production so as to ensure everything is working in a consistent fashion no matter the environment.

Running the Test Skeletons

When the RSpec gem is installed and you generate a new model, a new spec file is automatically generated for you. You can confirm this file is being created at the time the model is generated by watching for invocation of RSpec and confirmation of spec file creation in the model generation output. For instance, take a look at the output resulting from the `Location` model being generated:

```
$ rails generate model Location name:string
  invoke  active_record
  create    db/migrate/20140724130112_create_locations.rb
  create    app/models/location.rb
  invoke    rspec
  create      spec/models/location_spec.rb
  invoke      factory_girl
  create        spec/factories/locations.rb
```

Within this spec file you'll house tests related to the `Location` model. Open up the `spec/models/location_spec.rb` file and you'll find the following test skeleton:

```
require 'rails_helper'

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

The `RSpec.describe` block defines the behavior of the class, done by defining a series of tests. Currently there are no tests found in the block, but let's run the test anyway:

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

You should receive the following output:

```
*

Pending:
  Location add some examples to (or delete)
  /Users/wjgilmore/Software/dev.arcadenomad.com/spec/models/location_spec.rb
  # Not yet implemented
  # ./spec/models/location_spec.rb:4

Finished in 0.00044 seconds (files took 1.31 seconds to load)
1 example, 0 failures, 1 pending
```

We are able to successfully execute the placeholder test, as indicated by the lone asterisk. Let's update the spec file to include some basic tests.

Creating Your First Test

Let's start testing the `Location` model test by building a few `it` blocks, each of which describes a different model characteristic to be tested. Delete the `pending` line found in the `describe` block, replacing it with several `it` statements so the updated `location_spec` file looks like this:

```
require 'rails_helper'

RSpec.describe Location, :type => model do
  it "can be instantiated"
end
```

Run the test again and you'll be presented the following output:

```
$ rspec spec/models/location_spec.rb
*

Pending:
  Location can be instantiated
    # Not yet implemented
    # ./spec/models/location_spec.rb:4

Finished in 0.00129 seconds (files took 1.99 seconds to load)
1 example, 0 failures, 1 pending
```

The output indicates that the test is still pending, but this time we're seeing some indication of what the tests are intended to cover. So how do we actually test for instance whether an record of type `Location` can be instantiated? Let's revise the `it` statement, converting it into a block and creating an actual test:

```
RSpec.describe Location, :type => model do

  it "can be instantiated" do
    location = Location.new
    expect(location).to be_a Location
  end

end
```

I'll admit to getting ahead of things here since the `new` method has yet to be introduced. Even so, the purpose is likely obvious, as the name indicates it is used to create an object of type `Location`. Once created, this test uses RSpec's `expect` method in conjunction with `be_an_instance_of` to determine whether the newly created record is indeed of type `Location`.

Save the `Location` spec changes and run the test anew:

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

Finished in 0.00248 seconds (files took 1.31 seconds to load)
1 example, 0 failures
```

Aha! The output has changed. In particular, note how the output no longer states `1 pending`. This is because we've replaced the placeholder with an actual test, albeit a somewhat contrived one. Also, note the use of periods instead of asterisks to indicate the number of tests passed; this is because we're running actuals tests instead of placeholders.

Congratulations! You've created your first test, confirming the `Location` model has indeed be instantiated.

Let's try adding another test just to get the hang of the process. Below the existing test add the following code:

```
  it 'can be assigned the name of an arcade' do
    location = Location.new
    location.name = '16-Bit Bar'
    expect(location.name).to eq('16-Bit Bar')
  end
```

Save the changes and run the tests:

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

Finished in 0.03737 seconds (files took 2.07 seconds to load)
2 examples, 0 failures
```

If like me you prefer RSpec to be a tad more verbose when running tests, you can pass the `-fd` option (documentation format), prompting RSpec to list the tests that have been executed:

```
$ rspec -fd spec/models/location_spec.rb

Location
  can be instantiated
  can be assigned the name of an arcade

Finished in 0.00248 seconds (files took 1.31 seconds to load)
2 examples, 0 failures
```

These tests serve my goal of familiarizing you with the testing environment, but aren't particularly useful. Now that you have the general hang of things, let's turn our attention to a much more useful example. Recall the `address` virtual attribute we created in the earlier section, "Introducing Virtual Attributes":

```
 def address
  street + ' ' + city + ', ' + state + ' ' + zip
 end
```

We can confirm the `address` method returns the desired string using the following test:

```
it 'assembles a proper address virtual attribute' do

  location = Location.new
  location.name = '16-Bit Bar'
  location.street = '254 South Fourth Street'
  location.city = 'Columbus'
  location.state = 'Ohio'
  location.zip = '43215'

  expect(location.address).to eq('254 South Fourth Street Columbus, Ohio 43215')

end
```

Defining Fixtures Using FactoryGirl

You'll of course want to eliminate repetitive code within your tests, much of which occurs when setting up the model objects for testing purposes. For instance, presumably the `Location` model will soon grow to a certain level of complexity, requiring more than a dozen different tests, each of which requires you to create a new `Location` record and then manipulate its attributes. What if at some point in the project you added or deleted a `Location` model attribute? To account for such modifications, you would need to refactor all of the tests to account for the model changes. Obviously this is a situation you'd like to avoid, and fortunately there is an easy way to do so using the FactoryGirl gem. Presumably you installed the gem as directed at the beginning of this chapter; if not consider taking a moment to do so before reading on.

FactoryGirl removes the hassle of creating and configuring records for use within your tests by providing a facility for generating model *factories*. These factories are stored in a model-specific file found in `spec/factories/`. For instance if you open up `spec/factories/locations.rb` you'll find the following contents:

```
# Read about factories at https://github.com/thoughtbot/factory_girl

FactoryGirl.define do
  factory :location do
    name "MyString"
  end
end
```

Modify this file to look like this:

```
FactoryGirl.define do

  factory :location do
    name 'Pizza Works'
  end

end
```

After saving the file, return to the `Location` spec (`spec/models/location.rb`) and add the following test:

```
it "can be created using a factory" do
  location = FactoryGirl.build(:location)
  expect(location).to eq('Pizza Works')
end
```

Run the tests again and you should see the following output:

```
$ rspec -fd spec/models/location_spec.rb

Location
  can be instantiated
  has a valid factory
  can be assigned the name of an arcade

Finished in 0.04053 seconds (files took 2.05 seconds to load)
3 examples, 0 failures
```

Eliminating Redundancy Using Test Setups

While the factories eliminate the hassle of manually creating objects, we're still repeatedly generating these models in each of the tests created so far, violating the best practice of *staying DRY* ("Don't Repeat Yourself"). Is there a way to write this code only once, yet execute it before each test is run? Indeed there is, thanks to RSpec's `before(:each)` method. If the `before(:each)` method is defined within your test file, any code found within will execute before each and every test. Here's an example:

```
RSpec.describe Location, :type => :model do

  before(:each) do
    @location = FactoryGirl.build :location
  end

  it 'can be instantiated' do
    expect(@location).to be_an_instance_of(Location)
  end

  it 'has a default name of Pizza Works' do
    expect(@location.name).to eq('Pizza Works')
  end

end
```

Note the subtle but important difference regarding the returned `Location` object; the name is prepended with the at sign, just as would be the case were you creating a standard Ruby class and initializing an instance variable in the class constructor.

Linting Your Factories

Model factories are of little use if the model isn't properly initialized. For instance, as you'll learn in the next chapter, Rails provides you with a number of solutions for *validating* model data, such as ensuring a location name is never blank or a zip code contains exactly five digits. Neglecting to ensure your factory follows these requirements could produce unintended outcomes within your tests. You could ensure validity by adding a validation test to each of your model specs, such as the following:

```
it 'has a valid factory' do
  expect(Location.new).to be_valid
end
```

However, in doing so we're not exactly testing the application models but rather are testing the *factory*; isn't this something FactoryGirl should be automatically doing? Indeed it is, and with a simple configuration change you can leave it to FactoryGirl to do the validation testing for you. To do so, create a new directory named `support` inside of your project's `spec` directory. Next, create a new file named `factory_girl.rb` and add the following code to it:

```
RSpec.configure do |config|
  config.before(:suite) do
    begin
      DatabaseCleaner.start
      FactoryGirl.lint
    ensure
      DatabaseCleaner.clean
    end
  end
end
```

Save this file to the newly created `spec/support` directory. Once in place, the `FactoryGirl.lint` method will execute before each test suite, building each factory and then calling `valid?` to ensure the factories are valid. If `false` is returned for any factory, an exception of type `FactoryGirl::InvalidFactoryError` is raised and followed by a list of invalid factories. Here's a sample message:

```
The following factories are invalid: (FactoryGirl::InvalidFactoryError)

* location
```

Note the calls to `DatabaseCleaner.start` and `DatabaseCleaner.clean`. These are *not* part of FactoryGirl but are instead made available through another gem called Database Cleaner[30]. In certain cases calling `FactoryGirl.lint` will result in the creation of database records (notably when factories are configured to manage associations, something we'll discuss in Chapter 4), a behavior that will likely affect the results of your tests. The Database Cleaner gem will clean out those artifacts, ensuring each test begins with a clean environment. You might recall we installed this gem at the beginning of the chapter; if you opted not to and would like to use FactoryGirl's linting capability, be sure to return to that earlier section for installation instructions.

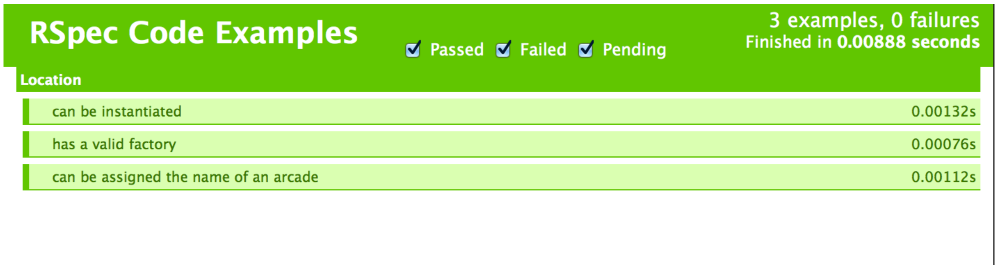
> In the days leading up to publication of the book I began encountering a strange error when executing `FactoryGirl.lint` that was causing tests to fail. I have for the moment commented out the call to `FactoryGirl.lint` in `factory_girl.rb`. Once the problem is resolved I'll post an explanation to the EasyActiveRecord.com blog.

Creating an HTML Test Report

The output format we've been using thus far is useful when you desire to receive immediate feedback regarding test results, however you might also wish to make these results available to other team members. One of the easiest ways to do so is by outputting the test results in HTML format. Recall how in earlier examples we specified documentation format using the `-fd` option. To switch to HTML format, use the `-fh` option, as demonstrated here:

```
$ rspec -fh spec/models/location_spec.rb > spec/report/index.html
```

Keep in mind the `spec\report` directory isn't created by default; you'll need to create it yourself or identify a different location if you'd like to create a definitive place for storing your test output. Example output is presented in the following screenshot.



RSpec Code Examples ☑ Passed ☑ Failed ☑ Pending 3 examples, 0 failures
Finished in **0.00888 seconds**

Location

can be instantiated 0.00132s

has a valid factory 0.00076s

can be assigned the name of an arcade 0.00112s

An Example RSpec HTML Report

[30] https://github.com/bmabey/database_cleaner

Useful Testing Resources

There are a number of fantastic online learning resources that you should definitely peruse:

- RSpec-Rails[31]: The RSpec Rails documentation is quite extensive, and definitely worth reading in order to better understand this powerful gem's testing reach.
- Better Specs[32]: This extensive resource covers dozens of RSpec best practices, and highlights numerous online and print learning resources.
- Everyday Rails Testing with RSpec[33]: Aaron Sumner has published a popular book on the topic. Currently this book covers Rails 4 and RSpec 2.1.4, however according to the Leanpub page readers will receive a free update when the updated edition is available.

Conclusion

I would imagine reading this introductory chapter felt like riding an informational whirlwind! Although a great many fundamental Active Record topics were covered, you'll repeatedly rely upon all of the features discussed, so be sure to read through the material a few times to make sure everything sinks in.

In the next chapter you'll learn how to populate your database with location and game data, save and manipulate data, and validate models to ensure the data remains consistent and error-free.

[31] https://relishapp.com/rspec/rspec-rails/docs

[32] http://betterspecs.org/

[33] https://leanpub.com/everydayrailsrspec

Chapter 2. Loading, Validating and Manipulating Data

In the interests of compiling the world's largest repository of arcade games, it's fair to say we'll be spending a lot of time inserting and editing game and location data. In the project's early stages this means batch inserting a bunch of data you may have previously compiled within a spreadsheet, and then as the project progresses you'll want to use forms, batch scripts and other mechanisms for continuing to add and manage the data. For instance you'll probably want to load some fairly boilerplate data such as a list of the 50 U.S. states (used to identify a particular location's state) and a list of 1980's video games (including their names, release dates, manufacturers, etc.) that you've already perhaps painstakingly amassed in a spreadsheet. Importing this sort of starter data is known as *seeding* the database. Further, you'll want to initialize other tables as the application evolves over time, perhaps inserting a set of categories into a newly created table used to segment arcades according to the *type* of location (laundromat, skating rink, etc.). I'll kick things off in this chapter by showing you how to easily seed initial data and subsequently insert new data as the need arises.

Next I'll present a detailed introduction to model validation. Data validation is crucial to any application's success, because a snazzy logo and sweet user interface will be of little consequence if the database is filled with erroneous information. Incomplete street addresses, redundant location entries, and missing phone numbers are going to irritate users, building little confidence in ArcadeNomad and therefore few reasons to continue using the application. Fortunately Rails offers a robust set of *validators* you can use to ensure any data meets your exacting specifications before being saved to the database. These specifications might be as simple as requiring a location to have a name, or as complex as ensuring an address includes a zip code comprised of exactly five integers. In this chapter I'll introduce you to these validators, showing you how to incorporate them into your models in order to wield maximum control over your application data. You'll also learn how to define and present custom error messages to the user should validation fail.

Following the introduction to validators I'll show you how to use Rails *callbacks* to automate the execution of code at various points along a model object's lifecycle. Among other things you can use callbacks to post-process user input prior to validation, notify an administrator of a newly added record, and even override the default behavior of model methods such as `destroy`.

We'll conclude the chapter with an in-depth introduction to the myriad ways in which you can create, edit and delete records, and additionally introduce an important new (to Rails 4) feature known as strong parameters..

Seeding and Updating Your Database

As is the case with so many of my personal projects, ArcadeNomad sprung to life as a simple YAML file ("YAML Ain't Markup Language"[34]) used to record various arcade game sightings made while milling around Columbus and traveling around the country for work or vacation. As the ArcadeNomad application sprung to life I at some point wanted to import this data and other boilerplate information (such as a list of U.S. states) into the application database without having to tediously insert it using a web form. Fortunately, a handy feature built into Rails greatly reduces the amount of work you'll need to do in order to initialize, or *seed*, your application data. You can easily import data sets into your project database using the `db/seeds.rb` file. Open this file and you'll find the following contents:

```
# This file should contain all the record creation needed to seed the database w\
ith its default values.
# The data can then be loaded with the rake db:seed (or created alongside the db\
 with db:setup).
#
# Examples:
#
#   cities = City.create([{ name: 'Chicago' }, { name: 'Copenhagen' }])
#   Mayor.create(name: 'Emanuel', city: cities.first)
```

As you can see from the examples found in the comments, you'll use the Ruby language (with Rails-infused ameliorations) to populate the tables. Of course, at this point in the book you presumably don't know what the `create` method does, however it doesn't take a leap of logic to conclude it does precisely what the name implies: it creates a new record! For instance, after creating the `State` model (used to normalize the arcade locations' state within the address) I added the following `create` method to the file (with some of the code removed; see this gist[35]) for a complete itemization of U.S. States):

```
State.create([
  { :name => 'Alabama', :abbreviation => 'AL'},
  { :name => 'Alaska', :abbreviation => 'AK'},
  ...
  { :name => 'West Virginia', :abbreviation => 'WV'},
  { :name => 'Wisconsin', :abbreviation => 'WI'},
  { :name => 'Wyoming', :abbreviation => 'WY'}
])
```

After saving the changes to `db/seeds.rb` you can load the data into your database using the following Rake task:

[34] http://en.wikipedia.org/wiki/YAML

[35] https://gist.github.com/wjgilmore/193544e26404e19dfaaf

```
$ rake db:seed
```

Once the Rake task has completed execution, provided the `db/seeds.rb` file is free of syntax errors and you've properly identified the model's attributes (`name` and `abbreviation` in this example), log into your development database and review the contents of the `states` table:

```
mysql> select * from states;
+----+---------------------+--------------+------------+------------+
| id | name                | abbreviation | created_at | updated_at |
+----+---------------------+--------------+------------+------------+
|  1 | Alabama             | AL           | ...        | ...        |
|  2 | Alaska              | AK           | ...        | ...        |
|    | ...                 |              |            |            |
| 49 | West Virginia       | WV           | ...        | ...        |
| 50 | Wisconsin           | WI           | ...        | ...        |
| 51 | Wyoming             | WY           | ...        | ...        |
+----+---------------------+--------------+------------+------------+
51 rows in set (0.00 sec)
```

Of course, you're not restricted to inserting data into just a single table; add as many model creation methods as you please to seed the database to the desired state. Furthermore, you're not even required to identify a table's contents using a statically-defined array as the above example demonstrated! Again, because the `db/seeds.rb` file is a standard Ruby script, you're free to take advantage of any Ruby library or available gem to retrieve and load your data, as well as manipulate the database contents to your liking. For instance, you might recall my mention of ArcadeNomad's humble beginnings as a simple YAML file. Rather than laboriously convert the YAML formatting into a static array I instead used Ruby's native YAML module to do the hard work for me. The YAML file (found in `db/seeds/games_list.yml`) used to create a catalog of the 1980's vintage arcade games I wanted to include in ArcadeNomad looks like this:

```
---
  - game: 1942
    year: 1984

  - game: After Burner II
    year: 1987

  - game: After Burner
    year: 1987

  ...
  - game: Xybots
    year: 1987

  - game: Zaxxon
    year: 1982

  - game: Zero Wing
    year: 1989
```

A simplified version of the code found in the ArcadeNomad code's `db/seeds.rb` file that is used to import this data is presented here:

```
games_data = YAML.load_file(Rails.root.join('db/seeds/games_list.yml'))

games_data.each do |game|

  game = Game.find_or_create_by(name, game['game'])
  game.release_date(game['year'])

  game.save

end
```

Note how in this case we're using the `find_or_create_by` to determine whether the game already exists. This allows you to repeatedly import this data should you for instance fix a mistaken release date within the `seeds.rb` file and want to import the change into the database without worrying about adding duplicates.

The sky is really the limit in terms of the different ways in which you can go about importing data. In recent projects I've had great success importing data via CSV, Excel spreadsheets, and web services.

Deleting Everything in the Database

Particularly in the early stages of development, you'll probably want to repeatedly revise the seed data. The most efficient way to do so is by making the desired changes within the `db/seeds.rb` file, and then import the data anew. When doing so you'll want to prevent duplicate entries from being inserted into the database. There are a few ways to avoid this undesirable outcome. You could use Rails' `find_or_create_by` method whenever you'd like to conceivably insert a new record into the database. Alternatively, you could use the `delete_all` method before beginning to insert data into a particular table, thereby deleting all of the table's records.

If you don't mind rebuilding the database schema, you can use the Rake task `db:reset`, which will drop and rebuild the tables using the migrations. You can use it in conjunction with `db:seed` like this:

```
$ rake db:reset db:seed
```

As a fourth option, you could simply delete all of the data in the database at the same time via a separate Rake task. I prefer this latter approach, as it saves the hassle of having to pay attention to the additional logic required using the former two approaches.

If you choose this latter approach, keep in mind you need to delete everything found in the database *except* for the data found in the `schema_migrations` table (this table was introduced in Chapter 1). There are several ways to go about this, however one of the easiest involves creating a Rake task to do it for you. I've pasted in the ArcadeNomad Rake file created for this purpose (found in `lib/tasks/utilities.rake`). It works by connecting to the database, and iterating over each table found in the database, truncating the table contents provided the table isn't named `schema_migrations`. If you're using SQLite, then be sure to comment the MySQL/PostgreSQL statement and uncomment the SQLite statement, as SQLite doesn't support a truncate command:

```
namespace :utilities do

  desc 'Clear database'
  task :clear_db => :environment do |t, args|

    ActiveRecord::Base.establish_connection
    ActiveRecord::Base.connection.tables.each do |table|

      next if table == 'schema_migrations'

      # MySQL / PostgreSQL
      ActiveRecord::Base.connection.execute("TRUNCATE #{table}")

      # SQLite
      # ActiveRecord::Base.connection.execute("DELETE FROM #{table}")

    end

  end

end
```

You can run this Rake task by executing the following command:

```
$ rake utilities:clear_db
```

With some simple changes to this task you could omit the truncation of tables other than `schema_migrations` should you want additional data to persist between truncations.

Adding Data Using Migrations

The `seeds.rb` file isn't necessarily intended to be continuously enlarged with creation statements over the course of the project; don't be afraid to simply delete any statements once you're satisfied with the data found in the database, replacing them with whatever new creation logic might relate to your latest schema enhancements. I think many developers overthink this simple concept, treating `seeds.rb` with kid gloves rather than constantly revising it to fit the project's current needs. After a time they forego using `seeds.rb` for what is perceived to be a "safer" approach: using migration files to insert or manipulate data. I don't agree with the idea of using migration files to manipulate data because such use supersedes the intended role. Even so, the practice is common enough and there are occasionally legitimate reasons for using migration files in this manner that I thought it worth including a section on the topic.

Suppose the list of arcades grows to the point that it makes sense to begin categorizing them according to the type of location the user would like to visit. Consider a user who needs to spend Sunday washing clothes and wishes to locate a laundromat with a few arcade games tucked into the corner. You might also classify locations as pool halls, skating rinks, airports, movie theaters, bars, and restaurants. After modifying the database schema to support location categorization, you'll want to load an initial set of category names. Although using the `seeds.rb` file is the preferable solution for adding these categories, you could also add them within the same migration file that creates the `categories` table. To do so you'll use `execute` to run SQL statements, Here's what the migration file might look like:

```
class CreateCategoriesTable < ActiveRecord::Migration
  def up

    create_table :categories do |t|
      t.string :name, :null => false
    end

    execute "INSERT INTO categories (name) VALUES('Laundromat')"
    execute "INSERT INTO categories (name) VALUES('Skating Rink')"
    execute "INSERT INTO categories (name) VALUES('Pool Hall')"
    execute "INSERT INTO categories (name) VALUES('Airport')"

  end

  def down
    drop_table :categories
  end

end
```

Callbacks

A *callback* is a bit of code configured in such a way that it will automatically execute at a predetermined time. Rails offers a number of callbacks capable of executing at certain points along the lifecycle of an Active Record object. For instance, you could configure a callback to execute before an object is saved, after an object has been validated, or even both before and after a record has been deleted. In fact, these are just a few of the points in which a callback can be triggered. Here's a complete list:

- `after_create`: Called after a new record has been created.
- `after_commit`: Called after the record save transaction has completed.
- `after_destroy`: Called after a record has been deleted.
- `after_find`: Called after a record has been retrieved via an Active Record query.
- `after_initialize`: Called after a model object has been instantiated.
- `after_rollback`: Called after a record save transaction has been rolled back.
- `after_save`: Called after a record has been saved to the database.
- `after_touch`: Called after a record has been "touched" via the `touch` method (used to update `_at` timestamp attributes without having to actually update another attribute).
- `after_update`: Called after a record has been updated.
- `after_validation`: Called after a model object has been validated.
- `around_create`: Provides the ability to execute code both before and after a record has been created.
- `around_destroy`: Provides the ability to execute code both before and after a record has been deleted.
- `around_save`: Provides the ability to execute code both before and after a record has been saved.
- `around_update`: Provides the ability to execute code both before and after a record has been updated.
- `before_create`: Called before a record has been created.
- `before_destroy`: Called before a record has been deleted.
- `before_save`: Called before a record has been saved.
- `before_update`: Called before a record has been updated.
- `before_validation`: Called before a model object has been validated.

It's important you pay close attention to the terminology used in these callback names. For instance, `after_commit` and `after_save` probably sound like they perform the same task, but there are indeed subtle but significant differences. Because Rails wraps its record saving procedure in a transaction, any logic associated with an `after_save` callback will be executed after the save but *inside* the transaction. The `after_commit` gives developers the ability to execute code *outside* of that transaction.

Similarly, the `before_create` and `before_save` callbacks sound as if they are identical in nature, but they are indeed different. The `before_create` callback will only execute in conjunction with *newly* created records, whereas `before_save` will execute prior to both the creation of new records and the update of existing records, in short executing whenever a record is being saved to the database (new or otherwise).

So, how do you actually configure a callback? The callbacks are defined within the desired model. Perhaps the easiest useful example involves writing to a custom logfile when a new record is created. For instance the following `after_create` callback will log a message to the currently active log file when a new location has been saved to the database:

```
class Location < ActiveRecord::Base

  after_create :log_location

  private

  def log_location
    logger.info "New location #{id} - #{name} created"
  end

end
```

Note the declaration of the `log_location` method as `private`. Declaring callback methods as `private` or `protected` is standard practice in accordance with sound object encapsulation principles.

After a new location has been added to the database, a message like the following will be logged:

```
New location 4 - Truck World created
```

If you wanted to be more proactive regarding receiving notifications, consider using Action Mailer[36] in conjunction with a callback to generate and send you an e-mail every time a new location is created.

Callbacks serve needs going well beyond custom logging or notifications. You might recall from Chapter 1 the example involving using a custom setter to strip all of the non-numeric characters from a telephone number. To refresh your memory I'll include the setter code here:

[36] http://guides.rubyonrails.org/action_mailer_basics.html

```
def telephone=(value)
  write_attribute(:telephone, value.gsub(/[^0-9]/i, ''))
end
```

While this approach works just fine, you could alternatively define a before_validation callback to update the telephone attribute in the same manner:

```
class Location < ActiveRecord::Base

  before_validation :normalize_telephone

  private

  def normalize_telephone
    telephone.gsub!(/[^0-9]/i, '')
  end

end
```

As a last example, callbacks could be used to modify the behavior of destructive methods such as destroy (introduced later in this chapter). For instance, many applications never actually delete data but instead mark the record in such a way so as to ensure it doesn't appear in future queries. For instance, you might never wish to actually delete ArcadeNomad locations but instead set a flag identifying them as having been removed from active listings. Yet it would still be nice to continue using the destroy method rather than resort to writing custom logic to achieve this effect. Using the before_destroy callback you can easily override the destroy method's behavior:

```
class Location < ActiveRecord::Base

  before_destroy :override_delete

  private

  def override_delete
    update_attribute(:deleted_at, Time.now)
    false
  end

end
```

In the override_delete callback we're using the update_attribute method to set the record's deleted_at attribute to the current date/time (the update_attribute method is introduced later in this chapter). We then return false to ensure the record is never actually deleted.

Of course, some of you may be wondering whether this approach is worthwhile given you would logically need to modify the application's queries to filter out any records having a non-null `deleted_at` attribute. Indeed you would, but of course you wouldn't be the only one to deal with this inconvenience meaning there are plenty of third-party gems capable of automating away this tedious task for you. Two of the more popular solutions are ActsAsParanoid[37] and Paranoia[38].

Useful Callback Resources

The material covered in this section is intended to provide you with a general but not exhaustive understanding of Rails' callbacks feature. Be sure to consult the Rails documentation[39] for a complete summary of capabilities. Also, consider perusing these valuable resources in order to gain a well-rounded understanding of the challenges associated with using callbacks:

- The Only Acceptable Use for Callbacks in Rails, Ever[40]: In this blog post, Jonathan Wallace decries the difficulties of debugging callbacks, and offers some solid advice regarding proper use.
- The Problem with Rails Callbacks[41]: Samuel Mullen offers some great advice about the challenges of managing callbacks and how to restructure your code to improve testability and overall organization.

Introducing Rails Validators

Invalid data such as blank location names, incomplete addresses, and mistyped phone numbers will not only infuriate users intent on finding the closest game of Space Invaders, but would also cascade into other areas of the application, hampering for instance the geocoder's ability to properly convert the location's address into the latitudinal and longitudinal coordinates necessary for performing tasks such as presenting those arcade locations found in proximity to the user. Fortunately you can take advantage of Active Record's *validation* features to ensure any data passed through your models meets your exacting specifications. Using a variety of validation methods, you can among other things ensure attributes are present, unique, are of a certain length, are numeric, follows the specifications of a regular expression, or meet more complex requirements through grouped and conditional validations.

In this section I'll provide an overview of Active Record's fundamental validator features, demonstrating how the `Location` and `Game` models can be enhanced to ensure all location and game data saved through the models matches your desired constraints, consequently emitting one or several errors should the provided data fall short of these expectations.

[37] https://github.com/technoweenie/acts_as_paranoid

[38] https://github.com/radar/paranoia

[39] http://edgeguides.rubyonrails.org/active_record_callbacks.html

[40] http://www.bignerdranch.com/blog/the-only-acceptable-use-for-callbacks-in-rails-ever/

[41] http://www.samuelmullen.com/2013/05/the-problem-with-rails-callbacks/

Creating Your First Validator

The presence of most attributes is generally non-negotiable; they should be included with every record. For instance the `Location` model's `name` attribute is logically always required, otherwise users wouldn't have a natural way to refer to the arcades. To ensure the `name` attribute is always present, open the `Location` model and add the following validators:

```
validates :name, presence: true
validates :description, presence: true
```

I like to add the validation definitions at the very top of my application models (open any of the ArcadeNomad models for an example), but this is just a matter of preference.

Validators can also be defined using an alternative `validates_x_of` syntax, like this:

```
validates_presence_of :name
validates_presence_of :description
```

In either case you can combine like validators into a single line, meaning the above four statements could also be written using the following consolidated variations, respectively:

```
validates :name, :description, presence: true
validates_presence_of :name, :description
```

While the behavior of these two syntactical variations is identical, the `validates` approach offers a slightly optimized syntax in that it allows for multiple validation types to be defined on a single line. I'll demonstrate this feature in the later section, "Combining Validations". Because of this, I'll use the `validates` approach throughout the remainder of the book and in the ArcadeNomad application.

After adding the new `name` and `description` presence validators to the `Location` model, confirm validation is working as you anticipate by opening up the Rails console and creating a new `Location` object:

```
$ rails console --sandbox
Loading development environment in sandbox (Rails 4.1.1)
Any modifications you make will be rolled back on exit
>> location = Location.new
#<Location id: nil, name: nil, description: nil, created_at: nil, \
updated_at: nil>
>> location.save
  (0.2ms)  BEGIN
  (0.2ms)  ROLLBACK
=> false
>>
```

A new record was not added to the locations database because when save was called, any validations associated with the Location model were first taken into consideration. Because a value was not assigned to the name property, the record could not be saved.

Checking Validity

You can check an object's validity before attempting to save it using the valid? method. The valid? method will return true if all of an object's validations pass and false otherwise. Should one or more validations fail, you can display their respective error messages by retrieving the errors collection. Let's continue using the Rails console session started in the previous example:

```
>> location.valid?
=> false
>> location.errors
=> #<ActiveModel::Errors:0x007ffd8a10d300 @base=#<Location id: nil, name: nil,
   description: nil, created_at: nil, updated_at: nil>,
   @messages={:name=>["can't be blank"], :description=>["can't be blank"]}>
>> location.errors.size
=> 2
```

The errors collection's messages attribute is a hash containing keys identifying the object's invalid properties and their associated error messages. These messages admittedly look at bit funny because they're incomplete sentences (can't be blank for both the invalid name and description properties because we assigned presence validators to these attributes). Rails will provide you with more user-friendly error messages when the errors collection's full_messages method is called:

```
>> location.errors.full_messages
=> ["Name can't be blank", "Description can't be blank"]
>>
```

While it may seem apparent that you should first call valid? and then call save should the former return true, you can actually just call save on the grounds that save is very likely returning false if and only if one or more validations have failed. Therefore for instance a simplified version of ArcadeNomad's Locations controller's new action would look like this:

```
@location = Location.new
@location.name = 'Pizza Palace'
@location.description = "Swatches and Jams are required attire at this
                         great 80's themed restaurant"

if @location.save
  redirect_to @location
else
  render 'new'
end
```

The `render 'new'` statement will only execute should the `@location` object's `save` method return `false`. Should this occur, you'll want to make sure the user is informed of the issue(s) which caused record persistence to fail. This is done by displaying any errors added to the `@location` object's `errors` collection. The code found in the view that is responsible for displaying this data usually looks quite similar to this:

```
<% if @location.errors.any? %>
  <ul>
  <% @location.errors.full_messages.each do |message| %>
    <li><%= message %></li>
  <% end %>
  </ul>
<% end %>
```

In a real-world application the `@location` object would be populated by data supplied within a web form; as mentioned this is a simplified version. In Chapter 5 I'll present several real-world examples that tie all of these concepts together.

Other Types of Validators

The presence validator is but one of many supported by Rails. In this section I'll introduce you to a variety of other commonly used validators, focusing on those validators that you're most likely to use within the majority of Rails applications. Be sure to check out all of the available validators within the Active Record Validations section[42] of the Rails documentation.

Validating Numericality

Whether its a zip code, arcade rating, or the year in which a video game first hit the market, you'll want to be sure the supplied value consists solely of integer or floating point values. You can do so using the `numericality` validator. For instance, suppose you want to ensure an arcade location's zip code consists solely of integers, you would define the validator like so:

[42] http://guides.rubyonrails.org/active_record_validations.html

```
validates :zip, numericality: true
```

This would however allow for values such as `45.0` and `3.14` to be supplied, because the numericality validator's default behavior is to allow both integers and floating point numbers. You can ensure only integer values are allowed by passing along the `only_integer` option:

```
validates :zip, numericality: { only_integer: true }
```

Numerous other options are available for constraining the attribute. For instance, suppose you added a feature that invited users to rate arcades on a scale of `0.0` to `5.0`. The `numericality` validator's default behavior is to accept any value provided it is an integer or floating point number, therefore you'll want to constrain the behavior using the `greater_than_or_equal_to` and `less_than_or_equal_to` options:

```
validates :rating, numericality: {
                   greater_than_or_equal_to: 0.0,
                   less_than_or_equal_to: 5.0
                   }
```

Validating Length

The `length` validator will constrain an attribute's allowable number of characters. You can use the `length` validator's supported options to constrain the length in a variety of ways. For instance, we can further constrain the previously mentioned `zip` attribute by limiting the length to exactly five characters using the `is` option:

```
validates :zip, length: { is: 5 }
```

You could use the `maximum` option to ensure an arcade review title doesn't surpass 30 characters:

```
validates :title, length: { maximum: 30 }
```

Similarly, you might want to ensure a user name is at least two characters:

```
validates :username, length: { minimum: 2 }
```

Finally, consider a scenario in which you wanted to constrain both the minimum and maximum lengths, such as might be desired for ensuring an arcade review contains an adequate number of characters to be informative but doesn't ramble on for pages. You could pass along both the `minimum` and `maximum` options, however a convenience option called `in` makes the task even easier:

```
validates :review, length: { in: 10..500 }
```

This ensures the supplied review is at least 10 but no greater than 500 characters in length.

Because of the `length` validator's various supported options, a number of different message-specific options are also available. As with the `numericality` validator you can use `message`:

```
validates :zip, length: { is: 5, message: 'The zip code must consist of
  exactly five digits.' }
```

Using the `too_short` and `too_long` options, you can tailor the message to specifically identify the nature of the problem when the supplied value either falls under or over the minimum or maximum number of allowable characters:

```
validates :review, length: {
  in: 10..500,
  too_short: 'An arcade review must consist of at least 10 characters',
  too_long: 'We appreciate your candor however please limit the review
             to 500 characters or less'
}
```

Validating a Custom Format

Sometimes it isn't enough to confirm a value is of a certain length or that it consists of solely integers. Consider a situation in which your ambitions to create the ultimate arcade aggregator become a tad unhinged and you conclude it is no longer suffice to merely request the first five digits of a zip code; You now want to require the user to enter all nine digits (also known by the United States Postal Service as the *ZIP+4 Code*). These zip codes would follow the format `XXXXX-XXXX`, in which each `X` is an integer value between `0` and `9`. For instance, Pizza Works' ZIP+4 code (as a teenager, Pizza Works was one of my favorite destinations for playing BurgerTime) is `44425-1422`. You can use the `format` validator to define a custom regular expression used to validate these sorts of specialized strings:

```
validates :zip_four,
  format: { with: '/\b[0-9]{5}-[0-9]{4}\b/' },
  message: 'The zip code must include all nine digits using the format 44425-142\
2!'
```

Creating a Custom Validator

If you plan on using a particular custom validator within multiple models or applications, such as the ZIP+4 validator presented in the above example, you can extract it to a separate file. As an example, create a directory within your application's `app` directory named `validators`, and in it create a file named `zip_validator.rb`, adding the following code to it:

```
class ZipValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\b[0-9]{5}-[0-9]{4}\b/
      record.errors[attribute] << (options[:message] || "is not a valid ZIP+4
        zip code.")
    end
  end
end
```

The name of the validator will be called simply `zip`, and so the class is named accordingly. It inherits from the `ActiveModel::EachValidator` validator, upon which all native Rails validators are built. This custom validator class only needs a single method named `validate_each` which accepts three parameters (`record`, `attribute`, and `value`). These parameters represent the record, attribute, and value of the attribute, respectively. If the `value` parameter does not match the provided regular expression (`/\b[0-9]{5}-[0-9]{4}\b/`) then the error message will be added to the record's `errors` array.

Next, you'll need to make the newly created directory (`app/validators`) available to your application's autoloader. Open up your application's `config/application.rb` file and add the following line:

```
config.autoload_paths += %W["#{config.root}/app/validators/"]
```

Save these changes and you can begin using the custom validator! For instance we could revise the earlier `zip_four` validator, removing the regular expression and using the new custom validator:

```
validates :zip, zip: true
```

Constraining Input to a Set of Predefined Values

Suppose a future version of ArcadeNomad included a store that sold coffee cups, t-shirts, and other branded items. T-shirt sizes come in four sizes, including small, medium, large, and extra large, and the t-shirt manufacturer requires this information be sent using the values `small`, `medium`, `large`, and `x-large`, respectively. You can constrain the `size` attribute to these four options using the `inclusion` validator:

```
validates :size,
  inclusion: {in: ['small', 'medium', 'large', 'x-large'],
  message: 'Please select a valid t-shirt size'}
```

You're not limited to defining supported values in an array; any enumerable object will do. For instance you could define the same validation using the `%w` (array of words) modifier:

```
validates :size,
  inclusion: {
    in: %w(small, medium, large, x-large),
    message: 'Please select a valid t-shirt size'
  }
```

Because any enumerable object is supported, it's possible to constrain input to an integer ranging between 13 and 100 without having to actually define each integer:

```
validates :age,
  inclusion: {
    in: 13..100,
    message: 'You must be over 12 years of age to join ArcadeNomad.'
  }
```

You are also able to *exclude* values using the `exclusion` validator, which works identically to `inclusion` except that the set of supplied values will determine what is disallowed rather than allowed. As an example, suppose you wanted to prevent users from selecting a username that could potentially be used to mislead others. You can create an array of restricted names and pass it into the `exclusion` validator, like so:

```
@disallowed_usernames = %w(admin root administrator moderator administrators
  boss bigboss owner)

validates :username,
  exclusion: {
    in: @disallowed_usernames,
    message: 'Please choose a unique username.'
  }
```

Confirming an Attribute

When registering for a new account, users are often asked to provide and then confirm a password. Although annoying, in an age where savvy users are choosing much longer and more complex passwords, it can be easy to mistype a password and foul up the registration process. This safeguard raises an interesting conundrum: all of the validators introduced so far are associated with an actual model attribute, but a password confirmation field would be used solely to determine whether the user is certain he's typed the password as intended. Recognizing this dilemma, the Rails developers created a special validator for expressly this purpose that will create a *virtual attribute* (see Chapter 1 for more about virtual attributes). For instance if the model attribute you'd like to validate in this fashion is called `password`, Rails will look for a form field named `password_confirmation`, comparing the two and ensuring identical values. You'll define the validator like this:

```
validates :password, confirmation: { message: 'The passwords do not match' }
validates :password_confirmation, presence: true
```

Note you need to define *two* validators when using this particular feature. The first and obvious validator determines which model attribute (`password`) will be confirmed by comparing its value with the `password_confirmation` virtual attribute. However, this validator will not trigger if the `password_confirmation` value is `nil`, meaning you also need to confirm the presence of the `password_confirmation` field. On an aside, while the `confirmation` validator is most commonly used in conjunction with passwords, it logically could be used in conjunction with any model attribute.

In Chapter 5 I'll show you how to integrate attribute validation into a web form.

Ensuring Uniqueness

You'll often want to ensure all attribute values are unique. For instance it wouldn't make any sense to list the manufacturer "Capcom" twice in the `manufacturers` table, therefore the `Manufacturer` model's `name` attribute is declared as being unique:

```
validates :name, uniqueness: true
```

Validating Dates

Believe it or not, Rails doesn't offer any native support for validating dates. This deficiency has always been a bit puzzling, given the prevalency in which users enter dates into all manner of web applications (birthdays, anniversaries, and travel dates just to name a few examples). There are however a few workarounds, several of which I'll introduce in this section.

> Early on in the development of this book I included information in this section about the validates_timeliness[43] gem, a fantastic and comprehensive solution for validating dates and times. However, as this book neared publication it became increasingly clear the gem was not being updated on a timely basis for Rails 4+, and was producing a deprecation warning pertinent to usage of syntax slated for removal in Rails 4.2. Given the likelihood the gem will be soon completely broken, I decided to remove coverage of this gem. However please do check the gem's GitHub page to determine whether maintenance has resumed, because when operational validates_timeliness really is an indispensable gem.

If you're working with just a year such as `1984` then the best solution is to use an `integer` column and validate the attribute using the `numericality` validator like so:

[43] https://github.com/adzap/validates_timeliness/

```
validates :release_year, numericality: { only_integer: true }
```

You can optionally constrain the allowable years to a specific range:

```
validates :release_date,
  numericality: {
    only_integer: true,
    greater_than_or_equal_to: 1970,
    less_than_or_equal_to: 1989,
    message: 'The release date must be between 1970 and 1989.'
  }
```

If you'd like to validate a date such as `2014-06-18`, `2014/06/18` or even `June 18, 2014`, check out the date_validator[44] gem, authored by Oriol Gual[45]. Install the gem by adding the following line to your project `Gemfile`:

```
gem `date_validator`
```

With the `date_validator` gem installed, you can create models that include attributes which use the underlying database's date data type (`date` in MySQL), and then validate those dates like so:

```
validates :release_date, date: true
```

Here's an example:

```
>> g = Game.new
>> g.name = 'Space Invaders IV'
>> g.release_date = '2014-06-18'
>> g.valid?
=> true
>> g.release_date = '2014/06/18'
>> g.valid?
=> true
>> g.release_date = 'June 18, 2014'
>> g.release_date.to_s
=> "2014-06-18"
>> g.valid?
=> true
>> g.release_date = 'Bozoqua 94, 2014'
>> g.valid?
=> false
```

You can also constrain the allowable dates to a specific range:

[44] https://github.com/codegram/date_validator

[45] https://github.com/oriolgual

```
validates :release_date,
  date: {
    after: Proc.new { Date.new(1970,01,01) },
    before: Proc.new { Date.new(1989,12,31) },
    message: 'Please select date between 01/01/1970 and 12/31/1989'
  }
```

You'll want to use `Proc.new`[46] to define your date range in order to prevent caching of the selected values. Of course if the values never change then this won't be an issue.

Validating Booleans

Boolean fields (a field with only two possible values: `true` or `false`) have a great many uses in web development, such as determining whether a new use would like to subscribe to the company newsletter or confirming whether a blog post should be made public. When incorporating a Boolean attribute into your model you'll want to ensure it's set to either `true` or `false`. To do so you'll use the `inclusion` validator (the `inclusion` validator was formally introduced earlier in the chapter):

```
validates_inclusion_of :newsletter, in: [true, false]
```

One gotcha involving Boolean validation is the mistaken assumption you can use the `presence` validator, on the grounds that a blank, or empty, value would be construed as "nothing" and therefore be treated as false. However, the `presence` validator uses Ruby's `blank?` method to determine whether an attribute has been assigned a value:

```
>> newsletter_value = false
>> newsletter_value.blank?
=> true
```

Allowing Blank and Nil Values

It's often the case that you only want to validate an attribute should a value be provided in the first place. That is to say, you'd like a blank value to be acceptable, but if a non-blank value is provided it should be validated against some set of restrictions. For instance, when adding a new arcade location you might wish to make providing a short description optional, but if one is provided you want to impose some length restrictions:

```
validates :review, length: { in: 10..500 }, allow_blank: true
```

If `nil` is an acceptable value for a particular attribute, you can use the `allow_nil` option:

[46] http://www.ruby-doc.org/core-2.1.2/Proc.html

```
validates :review, length: { in: 10..500 }, allow_nil: true
```

Confused about the difference between blank and nil? Check out the blog post, "The Difference Between Blank?, Nil?, and Empty?"[47].

Combining Validators

You'll often want to constrain a model attribute in a variety of ways, necessitating the use of multiple validators. For instance, when creating a new location you'll probably want to ensure that its name attribute is both present and unique. As you've already learned, this is easily accomplished using the `presence` and `uniqueness` validators:

```
validates :name, presence: { message: 'Please identify the arcade by name.' }
validates :name, uniqueness: { message: 'An arcade by this name already exists' }
```

You can optionally save a few keystrokes by combining validators like so:

```
validates :name,
  presence: { message: 'Please identify the arcade by name.' },
  uniqueness: { message: 'An arcade by this name already exists.' }
```

Conditional Validations

Sometimes you'll want to trigger a validation only if some other condition is met. For instance, suppose ArcadeNomad's popularity grows to the point that you decide to start offering some swag via an online store. Some products, such as leg warmers, will be available in multiple sizes (small, medium, and large) whereas size is irrelevant to other products, such as beverage coasters. You could configure some future `Product` class to only require the `size` attribute to be set if a `sizable?` method returns `true`:

[47] http://easyactiverecord.com/blog/2014/04/08/rails-syntax-tips-the-difference-between-blank-nil-and-empty/

```
class Product < ActiveRecord::Base

  validates :size, inclusion: { in: %w(small medium large) }, if: :sizable?

  def sizable?
    sizable == true
  end

end
```

In this example, the `size` attribute will only be evaluated to determine if it's set to `small`, `medium`, or `large` if the product's `sizable` attribute (presumably a Boolean) is set to `true`. If `sizable` is set to `false`, the validator will not execute.

There's actually quite a bit of flexibility built into Rails' conditional validation capabilities. See the Rails documentation[48] for a complete overview of what's available.

Testing Your Validations

When incorporating model validations into your application you'll also want to create tests to confirm your validations are properly configured. Mind you, the goal here is *not* to test Active Record's validation capabilities! Those features are constantly undergoing testing as part of the Rails project. Rather, you should use tests to confirm your model validations are configured in a manner that meets the desired requirements. For instance, if the `Location` model includes tests ensuring the name is present, the zip code contains exactly five digits, and the description consists of between 50 and 100 characters, then you'll want to write tests to confirm these validations are configured to meet these exacting needs. With that said, let's consider a few examples.

After attaching a `presence` validator to the `Location` model's `name` attribute, we can write a test to make sure it's always configured as desired:

```
before(:each) do
  @location = FactoryGirl.build :location
end

...

it 'is invalid without a name' do
  expect(@location).to_not be_valid
end
```

Run the test again and you will see that it fails, because this time the `Location` record *is* valid:

[48] http://edgeguides.rubyonrails.org/active_record_validations.html#conditional-validation

```
Location
  is invalid without a name (FAILED - 1)

Failures:

  1) Location is invalid without a name
     Failure/Error: expect(@location).to_not be_valid
      expected #<Location ...> not to be valid

Finished in 0.63689 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/models/location_spec.rb:4 # Location is invalid without a name
```

Next, modify the test to set the `Location` object's `name` attribute to empty. Notice how we set the `name` attribute following creation of the factory in order to test the validator:

```
it "is invalid without a name" do

  @location.name = ''
  expect(@location).to_not be_valid

end
```

Run the test again and it will pass. What about a slightly more complicated test, such as whether the zip code validator is properly configured? You might recall we specified that the `zip` attribute must consist of exactly five integers. Let's create a few tests to confirm the validator is properly written:

```
it 'is invalid when the zip code does not consist of five integers' do

  @location.zip = '4320'
  expect(@location).to_not be_valid

end

it 'is invalid when the zip code does not consist of only integers' do

  @location.zip = '1234g'
  expect(@location).to_not be_valid

end
```

What about ensuring a location of the same name can't be saved to the database? There are a few ways you can go about testing this particular requirement, one of which follows:

```
it 'is invalid if name not unique' do

  @location.save

  @location_duplicate = FactoryGirl.build :location

  expect(@location_duplicate.save).to be_falsey

end
```

The `be_falsey` matcher is relatively new to RSpec, passing if the object is `nil` or `false`. Prior to RSpec 3.0 this matcher was called `be_false`. Similarly, `be_truthy` was previously called `be_true`, and passes if the object is not `nil` or anything else but `false`.

Try creating a few other validation tests to confirm your models are properly configured. If you purchased the ArcadeNomad project code, be sure to check out the various model specs for other examples.

Creating, Updating, and Deleting Records

Inserting data using the `db/seeds.rb` file and via migrations is useful for administrative purposes, however for applications like ArcadeNomad most data will be inserted and updated via the web interface. Earlier in the chapter you were already tangentially introduced to Active Record's `create` method, and indeed while `create` is commonly used for saving data (I'll formally introduce the method in this section), there are plenty of other ways in which records can be created. In fact, Rails' flexibility in this regards is often cause for some confusion among newbies, and so my hope is this section will go a long way towards eliminating any uncertainty you might otherwise encounter.

And of course, inserting new records is only one of several commonplace tasks your application will likely require; you'll also need to update record data and even occasionally delete records. In this section we'll go into great detail regarding how Rails facilitates these crucial operations.

Let's begin with what is perhaps the easiest example in which you create a new object of type `Location` and subsequently save it to the database. For this and many of the examples found in this section we'll use a simplified version of the actual ArcadeNomad `locations` table, presented here:

```
mysql> describe locations;
+-------------+----------------+------+-----+--------------+----------------+
| Field       | Type           | Null | Key | Default      | Extra          |
+-------------+----------------+------+-----+--------------+----------------+
| id          | int(11)        | NO   | PRI | NULL         | auto_increment |
| name        | varchar(255)   | YES  | MUL | NULL         |                |
| description | text           | YES  |     | NULL         |                |
| street      | varchar(255)   | YES  |     | NULL         |                |
| city        | varchar(255)   | YES  |     | NULL         |                |
| state       | varchar(255)   | YES  |     | NULL         |                |
| zip         | varchar(255)   | YES  |     | NULL         |                |
| created_at  | datetime       | YES  |     | NULL         |                |
| updated_at  | datetime       | YES  |     | NULL         |                |
+-------------+----------------+------+-----+--------------+----------------+
9 rows in set (0.00 sec)
```

You might recall from the last chapter that Active Record will handle persisting the `id`, `created_at` and `updated_at` fields for you, leaving us to deal with the `name`, `description`, `street`, `city`, `state`, and `zip` fields. You can save a new location to the database by instantiating the `Location` class, assigning the attributes, and then calling the `save` method, as demonstrated within the Rails console:

```
>> location = Location.new
>> location.name = "Dave & Buster's Hilliard"
>> location.description = "Hilliard location of the popular chain."
>> location.street = "3665 Park Mill Run Dr"
>> location.city = "Hilliard"
>> location.state = "Ohio"
>> location.zip = "43026"
>> location.save
```

While this example is ideal for demonstrating the `save` method's basic behavior, in practice you'll want to check the method's return value because `save` executes model validations before attempting to save the record to the database. If the validations fail, `save` will return `false`, meaning a simple conditional statement will do for confirming the outcome. Let's revise the above example to account for a potential persistence failure:

```
>> location = Location.new
>> location.name = "Dave & Buster's Hilliard"
>> location.description = "Hilliard location of the popular chain."
>> location.street = "3665 Park Mill Run Dr"
>> location.city = "Hilliard"
>> location.state = "Ohio"
>> location.zip = "43026"
>> if location.save
>>  puts "Save successful!"
>> else
?>  puts "Save failed!"
>> end
```

As you'll see in later examples where we integrate this logic into a Rails application controller, the above pattern is rather typical.

Although the code in the previous two examples is quite readable, assigning attributes in this manner has always struck me as rather tedious. You can eliminate a few keystrokes by passing parameters into the `new` constructor, like so:

```
>> location = Location.new(name: "Dave & Buster's Hilliard",
?> description: "Hilliard location of the popular chain",
?> street: "3665 Park Mill Run Dr",
?> city: "Columbus", state: "Ohio", zip: "43016")
>> location.save
```

I've never been a fan of this particular approach, because it just looks messy. You could clean things up a bit using *block initialization*, as demonstrated here:

```
location = Location.new do |l|
  l.name = "Dave & Buster's Hilliard"
  l.description = "Hilliard location of the popular chain."
  l.street = "3665 Park Mill Run Dr"
  l.city = "Hilliard"
  l.state = "Ohio"
  l.zip = "43026"
end

location.save
```

You can also pass a hash into the `new` constructor as demonstrated here:

```
>> new_location = {}
>> new_location[:name] = "Dave & Buster's Hilliard"
>> new_location[:description] = "Hilliard location of the popular chain"
>> new_location[:street] = "3665 Park Mill Run Dr"
>> new_location[:city] = "Hilliard"
>> new_location[:zip] = "43016"
>> location = Location.new(new_location)
```

Keep in mind the above variations all ultimately accomplish the same goal of creating a new record; whether you choose to separately assign each attribute, pass attributes in via the `new` constructor, or use block initialization is entirely a matter of preference.

The Difference Between save and save!

The `save!` method is a variation of the `save` method that behaves identically to `save` in every way except that it will throw an `ActiveRecord::RecordInvalid` exception. While you might be inclined to presume `save!` is therefore preferred because you could eliminate the conditional logic and rescue the exception, I urge you to take a moment to read the fantastic blog post written by Jared Carroll titled "save bang your head, active record will drive you mad"[49]. In this post he makes a great argument for why persistence failure is actually *expected* rather than unexpected, meaning exception handling in this context is actually not the best practice.

The `save` and `save!` methods are just one of many such variations; for instance `create` and `create!` methods also exist to serve the same purpose.

Creating Records with the Create Method

The `create` method saves you a few keystrokes when creating a new record because it bundles the behavior of `new` and `save` into a single step:

```
>> new_location = {:name => "Dave & Buster's Hilliard", ..., :zip => "43016"}
>> location = Location.create(new_location)
```

There is however a very important distinction between `create` and `save`: the `save` method will return `true` or `false` depending on whether the record was successfully saved, while `create` will return a model object regardless of outcome! Therefore attempting to use `create` in conjunction with a conditional statement is likely to have undesirable consequences, meaning you should probably stick with using `new` and `save` for most purposes unless you're certain record creation is going to be successful (I typically use `create` within my tests, for instance).

[49] http://robots.thoughtbot.com/save-bang-your-head-active-record-will-drive-you-mad

The convenience of simply passing a hash into the `create` and `new` methods is undeniable, however such capabilities can be quite dangerous if you're simply passing a hash containing user input into the mass-assignment method. Fortunately, Rails has long offered a safeguard for preventing a third-party from misusing this syntax. I'll introduce this safeguard in the later section, "Introducing Strong Parameters".

Creating But Not Saving Records with the Build Method

If you want to create *but not save* an object, you can use the `build` method:

```
>> new_location = {:name => "Dave & Buster's Hilliard", ..., :zip => "43016"}
>> location = Location.build(new_location)
```

Updating Records

The `save` method isn't used solely for saving new records; you can also use it to update an existing record's attributes. Suppose for instance you'd like to improve an existing arcade's description. You could retrieve the arcade using `find` and then modify the retrieved record's `description` attribute:

```
>> location = Location.find(3)
  Location Load (0.9ms)  SELECT  `locations`.* FROM `locations`
    WHERE `locations`.`id` = 3 LIMIT 1
=> #<Location id: 3, name: "Ethyl & Tank", description: "University restaurant,
    bar and arcade",
    street: "19 13th Avenue", city: "Columbus", zip: "43201",
    created_at: "2014-06-04 20:35:02", updated_at: "2014-06-04 20:35:02">
>> location.description = "The Ohio State University's newest and hottest bar!"
>> location.save
```

As with creating a new record, `save` will first confirm any model validations pass before saving the modified record to the database, meaning in practice you'll want to use a conditional to determine the outcome.

Mass Assignment with the update Method

If you'd like to simultaneously update multiple attributes you can use the `update` method (introduced in Rails 4), passing in a hash containing the desired attributes and their new values as demonstrated here:

```
>> location = Location.find(3)
>> updated_attributes = {:street => '1234 Jump Street', :city => 'Plain City',
    :zip => '43064'}
>> location.update(updated_attributes)
```

The `update` method validates the record before saving it to the database, returning `false` if the object is invalid.

What Happened to the update_attributes Method?

Rails 2 and 3 users are likely familiar with a mass-assignment method named `update_attributes`. This method's implementation was removed in Rails 4, and now serves as an alias for `update`.

As with the `create` method, you should take care when incorporating `update` into your Rails applications because while mass-assignment method offer a certain level of convenience to the developer, they have the potential to be quite dangerous if you're simply passing a hash containing user input into the mass-assignment method. Fortunately, Rails has long offered a safeguard for preventing a third-party from misusing this syntax. I'll introduce this safeguard in the later section, "Introducing Strong Parameters".

Introducing the update_columns and update_column Methods

Two other methods are available for updating record attributes: `update_columns` and `update_column`. The `update_columns` attribute is fast because it skips all validations and callbacks, while allowing you to easily update multiple columns:

```
>> location = Location.find(3)
>> location.update_columns(:city => 'Dublin', :zip => '43016')
>> l.update_columns(:city => 'Dublin', :zip => '43016')
  SQL (3.4ms)  UPDATE `locations` SET `locations`.`city` = 'Dublin',
  `locations`.`zip` = '43016' WHERE `locations`.`id` = 1
=> true
```

Note how the call to `update_columns` will trigger the database operation, negating the need to explicitly call `save`.

If you only need to update a single column, consider using the `update_column` method:

```
>> location = Location.find(3)
...
>> location.update_column(:description, "The Ohio State University's newest
    and hottest bar!")
SQL (0.6ms)  UPDATE `locations` SET `locations`.`description` =
'The Ohio State University\'s newest and hottest bar!' WHERE `locations`.`id` = 1
  => true
```

The update_column method works identically to update_columns, skipping validations and callbacks.

Creating a Record if It Doesn’t Already Exist

It is often useful to consult the database to determine whether a record associated with some specific attribute already exists, and if not, create the record. You can do this using the find_or_create_by method. For instance we can determine whether a manufacturer named “Capcom” already exists in the manufacturers table, and if not, create it:

```
>> m = Manufacturer.find_or_create_by(name: 'Capcom')
Manufacturer Load (0.4ms)  SELECT  `manufacturers`.* FROM `manufacturers`
WHERE `manufacturers`.`name` = 'Capcom' LIMIT 1
=> #<Manufacturer id: 1, name: "Capcom", ...>
```

In this example the record has indeed been found and returned. Now what about the little-known manufacturer “Atarcomidway”? Let’s see if it already exists, and if not, create it:

```
> m = Manufacturer.find_or_create_by(name: 'Atarcomidway')
Manufacturer Load (0.4ms)  SELECT  `manufacturers`.* FROM `manufacturers`
WHERE `manufacturers`.`name` = 'Atarcomidway' LIMIT 1
 (0.4ms)  BEGIN
   Manufacturer Exists (0.4ms)  SELECT  1 AS one FROM `manufacturers`
   WHERE `manufacturers`.`name` = BINARY 'Atarcomidway' LIMIT 1
     SQL (6.2ms)  INSERT INTO `manufacturers` (`created_at`, `name`,
     `updated_at`) VALUES ('2014-07-29 18:41:39',
     'Atarcomidway', '2014-07-29 18:41:39')
 (3.1ms)  COMMIT
=> #<Manufacturer id: 30, name: "Atarcomidway", ...>
```

You can optionally pass along multiple attributes. For instance you could determine whether a particular location name already existed in a given zip code and if not, create it:

```
> g = Game.find_or_create_by(name: "Pacman's Pizza", zip: 43016)
```

Creating and Updating Models within Your Rails Application

In earlier examples involving the `save` method I used `puts` to provide feedback regarding whether the record was successfully saved. However, when integrating record creation features into an actual Rails application, you'll need a different approach for informing the user, notably using Rails' convenient flash hash[50] and redirection to keep the user in the loop.

As mentioned earlier in this section, when saving a record from within a Rails application, you'll typically use Rails' flash hash[51] and redirection to keep users informed regarding the outcome. However, there are a few other noteworthy matters pertaining to this process, and so I thought it worth devoting a section to the topic. For starters, new records representing entities such as arcades or games are typically created using a *web form*. Web forms are undoubtedly a crucial part of any web application, and so in Chapter 5 you'll find an entire chapter devoted to the topic, with extensive coverage devoted to how Rails can make your life easier in regards to both generating forms and processing form data. Therefore rather than redundantly introduce that aspect of the record saving process here I'd instead like to focus on what happens *after* that form is submitted.

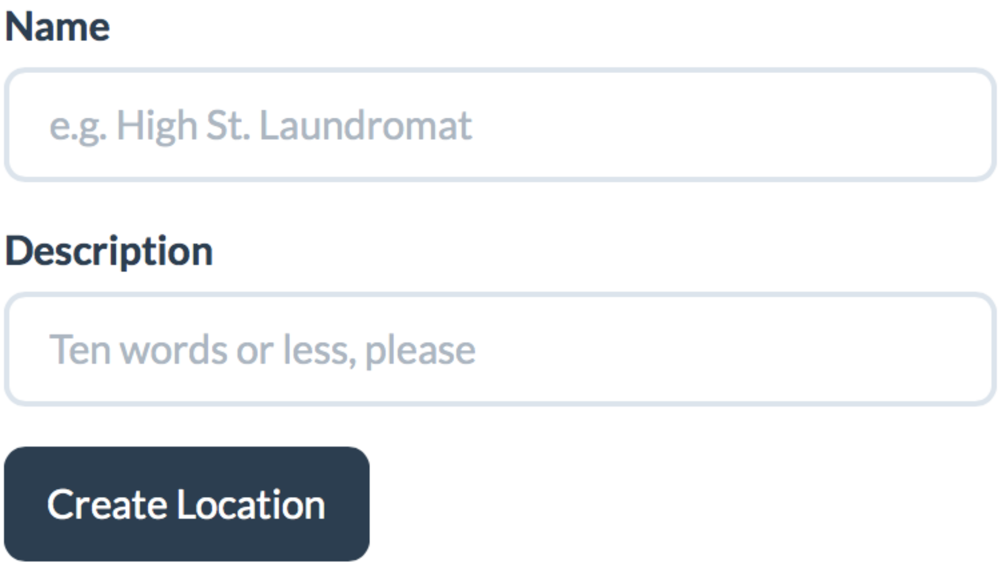
When the form is submitted to the destination URL (defined by the form's `action` attribute), the fields defined within the form and their associated values will be passed to the action associated with the destination URL and made available via the `params` hash. Let's use a very simple form as an example:

```
<form method="post" action="/locations/create">
  <div>
    <label>Name</label>
    <input type="text" name="location[name]"
           placeholder="e.g. High St. Laundromat">
  </div>
  <div>
    <label>Description</label>
    <input type="text" name="location[description]"
           placeholder="Ten words or less, please">
  </div>
  <div>
    <button type="submit">Create Location</button>
  </div>
</form>
```

Rendered within the browser, this form might look like this:

[50] http://guides.rubyonrails.org/action_controller_overview.html#the-flash

[51] http://guides.rubyonrails.org/action_controller_overview.html#the-flash



A simple location creation form

Although for the purposes of this section the above example is perfectly suitable, do *not* forge ahead and start creating web forms using the above example as your guide until after having read Chapter 5. In Chapter 5 I'll show you how to take advantage of native Rails features to generate forms such as that presented above.

This form contains two text fields named `location[name]` and `location[description]`, respectively. When the user submits the form, the names of these fields and the values assigned to them will be bundled into a hash named `params`. If you were to output the contents of this hash within the Rails console it would look like this:

```
>> params[:location]
=> {:name=>"Pizza Works", :description=>"Best pizza in Hubbard, Ohio!"}
```

You can then refer to this `params` hash when creating the new `Location` record. Here's what the action associated with the destination URL might look like:

```
def create

  @location = Location.new(location_params])

  if location.save
    flash[:notice] = 'New location created!'
    redirect_to location_path(@location.id)
  else
    render :action => 'new'
  end

end
```

But wait a second? Where is the `params` hash, and why is `location_params` instead being passed into the model constructor? We're not passing the `params` hash directly into the `new` method because doing so would be a security hazard. Instead, we're passing the return value of a method named `location_params` which has been created to filter out any attributes not intended for mass assignment. Exactly what is going on here will become abundantly clear in the next section, "Introducing Strong Parameters", so bear with me for just a moment. After creating and populating the `Location` object, we'll try to save it. If successful, the flash hash is populated with the message `New location created!`, and the user is redirected to the path defined by the `location_path` route. If attempts to save the record failed, the flash hash is populated with the message `Could not create new location` and the `new` action is rendered.

Introducing Strong Parameters

The Rails team has always been careful to take precautions that help to prevent malicious attackers from compromising an application's data. One of the most visible historical protections involved using the `attr_accessible` method within a model to identify which model attributes could be passed into methods like `new` and `update_attributes` for mass assignment. For instance when using Rails 3 if you only wanted to allow the `Location` model's `name`, `street`, and `city` attributes to be updatable via mass assignment you would define `attr_accessible` like so:

```
class Location < ActiveRecord::Base

  attr_accessible :name, :street, :city

  ...

end
```

By giving the developer control over which parameters were *whitelisted* for such purposes, the presumption was that data could be better protected by eliminating the possibility an attacker could modify a sensitive attribute by injecting it into a form body. This worked great but had a significant drawback, because `attr_accessible` was an all-or-nothing proposition. The developer couldn't for instance change the model requirements when working within an administration console, because `attr_accessible` could only be defined once. This changed with Rails 4, thanks to a new approach for managing mass assignment behavior. Known as *strong parameters*, the task of defining which parameters are available for mass assignment has been moved out of the model and into the controllers, allowing developers to define mass assignment behavior according to action.

Consider a simplified version of the `Game` model that consisted of the attributes `name`, `description` and `active`, the latter determining whether the game was displayed on the site. If the site were community-driven you are probably fine with `name` and `description` being updated but want to restrict any updates to `active` to administrators. Therefore you would use the strong parameters approach to define a method in the (presumably) `Games` controller that looks like this:

```
class GamesController < ApplicationController

  ...

  def game_params

    params.require(:game).permit(:name, :description)

  end
```

This method would then be passed into the `update` action's `update` method:

```
def update

  @game = Game.find(params[:id])

  if @game.update(game_params)
    ...
  end

end
```

Meanwhile, in an administration console you logically would want the ability to update the `active` attribute. In the appropriate administration controller you could define another method that looks like this:

```
def game_params

  params.require(:game).permit(:name, :description, :active)

end
```

This would give the administrator the ability to update the `active` attribute right alongside `name` and `description` using mass-assignment.

If one of your attributes accepts an array (such as is the case when working with certain types of associations; more on this topic in Chapter 4), you'll need to define that attribute a bit differently in the `permit` method. For instance the following example is a variation of the same strong parameters method used in ArcadeNomad's `Location` controller, because we need to pass along an array of game IDs when creating a new location:

```
def location_params

  params.require(:location).permit(:name, ..., :category_id, :game_ids => [])

end
```

This last example will make much more sense after you've read Chapter 4, so don't worry about it too much right now if you're not already familiar with Active Record associations.

Deleting and Destroying Records

The final topic we'll discuss in this chapter is how to remove records from the database. You might wonder why I titled the section "Deleting and Destroying Records", since one could conclude "deleting" and "destroying" are the same thing however in regards to Rails there is indeed a rather significant difference between these two terms..

When *deleting* a record you'll remove it from the database without regards to validations or callbacks. You can delete a record in a variety of ways; for instance the following three commands will all accomplish the same goal:

```
>> Manufacturer.delete(30)
>> Manufacturer.delete(Manufacturer.find(30))
>> Manufacturer.find(30).delete
```

You can also delete records based on other attributes using the `delete_all` method. For instance to delete all locations found in a city named `Columbus` you'll pass the name attribute into the `delete` method:

```
>> Location.delete_all(city: 'Columbus')
```

The `delete` and `delete_all` methods are convenient because they are fast, but should only ever be used when you are certain no side effects will occur due to the disregard for callbacks or validations. If you require the callbacks and validations to execute prior to record removal (and in most cases you will), you'll want to use `destroy` and `destroy_all`. These two methods work identically to `delete` and `delete_all`, respectively, except they ensure the execution of any other relevant domain logic as part of the record removal process.

Conclusion

Despite being only two chapters in we've already covered a tremendous amount of territory! In the next chapter we'll forge even further ahead, reviewing just about every conceivable approach you can use to query your database using Active Record. Onwards!