

A Hands On Introduction Using
a Real-World Project

SOLD TO THE FINE
sydpolk@gmail.com



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/easyactiveresord>

This version was published on 2014-12-09



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.

©2013 - 2014 W. Jason Gilmore

For Carli, my bride today and forever.

Contents

Introduction	1
Introducing Active Record	2
About this Book	5
Introducing the ArcadeNomad Theme Project	8
About the Author	8
Errata and Suggestions	9
 Chapter 1. Introducing Models, Schemas and Migrations	 10
Creating the ArcadeNomad Project	10
Installing the RSpec, FactoryGirl and Database Cleaner Gems	16
Introducing Rails Models	17
Rails Models are Plain Old Ruby Objects	21
Introducing Migrations	27
Testing Your Models	37
Conclusion	47
 Chapter 2. Loading, Validating and Manipulating Data	 48
Seeding and Updating Your Database	49
Callbacks	54
Introducing Rails Validators	57
Creating, Updating, and Deleting Records	71
Deleting and Destroying Records	82
Conclusion	83
 Chapter 3. Querying the Database	 84
Finding Data	84
Paginating Results	100
Creating a Prettier URL with FriendlyId	104
Introducing Scopes	108
Conclusion	111
 Chapter 4. Introducing Associations	 112
Normalizing Your Database	112
Introducing Associations	113

CONTENTS

Eager Loading of Associations	160
Polymorphic Models	162
Conclusion	164
Chapter 5. Mastering Web Forms	165
Web Form Fundamentals	165
Rails Routing and Forms	169
Creating a Simple Contact Form	172
Managing Your Application Data	185
Useful Forms-Related Gems	206
Conclusion	212
Chapter 6. Debugging and Optimizing Your Application	213
Debugging Your Application	213
Optimizing Your Application	223
Conclusion	230
Chapter 7. Integrating User Accounts with Devise	231
Installing Devise	231
Creating the User Account Model	232
Adding User Registration Capabilities	234
Adding User Sign In and Sign Out Capabilities	236
Adding User Account Management Capabilities	237
Adding Password Recovery Capabilities	238
Restricting Access to Authenticated Users	239
Commonplace Devise Customizations	239
Conclusion	242
Chapter 8. Powerful Active Record Plugins	243
Tagging Your Content Using the Acts As Taggable On Gem	243
Cleaning Up Your Controllers with the Decent Exposure Gem	249
Geocoding Your Models	252
Summary	256

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](http://www.rubyonrails.org/)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.



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](http://en.wikipedia.org/wiki/Haversine_formula)⁴. Fortunately thanks to Ruby's rich gem ecosystem we'll rely on the [Geocoder](https://github.com/alexreisner/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:

⁴http://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](http://www.railstutorial.org/)"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](http://rspec.info)⁷ and [factory_girl](https://github.com/thoughtbot/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>

⁸https://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 unglamorous 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](http://arcadenomad.com)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](http://getbootstrap.com/)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](http://www.postgresql.org/docs/)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 than 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](http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Classes)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.

²³http://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 entered 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 defined 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

One 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 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](http://guides.rubyonrails.org/migrations.html)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 `modelConfigurationSetting`. 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](http://www.codinghorror.com/blog/2007/03/primary-keys-ids-versus-guids.html)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](http://rspec.info/)²⁷, and in this section I'll show you how to use RSpec and another great gem called [FactoryGirl](https://github.com/thoughtbot/factory_girl)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](http://www.sqlite.org/)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/>

²⁸https://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](https://github.com/bmabey/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		3 examples, 0 failures Finished in 0.00888 seconds
<input checked="" type="checkbox"/> Passed <input checked="" type="checkbox"/> Failed <input checked="" type="checkbox"/> Pending		
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

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

Useful Testing Resources

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

- [RSpec-Rails](https://relishapp.com/rspec/rspec-rails/docs)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](http://betterspecs.org/)³²: This extensive resource covers dozens of RSpec best practices, and highlights numerous online and print learning resources.
- [Everyday Rails Testing with RSpec](https://leanpub.com/everydayrailsrspec)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

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](http://guides.rubyonrails.org/action_mailer_basics.html)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:

³⁶http://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.

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

³⁸<https://github.com/radar/paranoia>

³⁹http://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 a 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](http://guides.rubyonrails.org/active_record_validations.html)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:

⁴²http://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 prevalence 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](https://github.com/adzap/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:

⁴³https://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](https://github.com/codegram/date_validator)⁴⁴ gem, authored by [Oriol Gual](https://github.com/oriolgual)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:

⁴⁴https://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?”](http://easyactiverecord.com/blog/2014/04/08/rails-syntax-tips-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](http://edgeguides.rubyonrails.org/active_record_validations.html#conditional-validation)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 ActiveRecord'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:

⁴⁸http://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 ActiveRecord'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](http://robots.thoughtbot.com/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:

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

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

Name

e.g. High St. Laundromat

Description

Ten words or less, please

Create Location

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

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!

Chapter 3. Querying the Database

While talking about Active Record's many exciting features makes for great discussion, the reality is you're going to spend the majority of your time immersed in the seemingly pedestrian but actually often complicated act of retrieving data from the database. In fact, the majority of this book is devoted to helping you navigate the many ins and outs of data retrieval. We'll kick things off in this chapter with a wide-ranging introduction to fundamental Active Record query features, showing you how to retrieve, filter, sort, count and group data. Along the way I'll take plenty of opportunities to highlight many of the gotchas often encountered by new Active Record users.

Finding Data

Most Rails queries are very straightforward in that they'll simply involve retrieving a record based on its primary key or some other filter, while others require more sophisticated approaches involving multiple parameters, complex sorting, and table joins. Fortunately Rails offers an incredibly rich set of methods for querying data in a variety of fashions. In this section I'll show you the many ways in which data can be retrieved from your application's database.

Retrieving All Records

Perhaps the easiest Active Record query involves retrieving all of a table's records using the `all` method. The following example will retrieve all of the arcade locations:

```
locations = Location.all
```

The `locations` variable is an instance of `ActiveRecord::Relation`, which is among other things iterable. This means you can loop over the records using standard Ruby syntax. Fire up the Rails console and try it for yourself:

```
>> locations = Location.all
    Location Load (1.1ms)  SELECT `locations`.* FROM `locations`
>> locations.each do |location|
  ?> p location.name
>> end
Rusty Bucket
BW3s Dublin
Pizza Works
```

In the typical Rails application you'll want to iterate over the results in a view using a bit of presentational markup, as demonstrated here:

```
<ul>
  <% @locations.each do |location| %>
    <li><%= location.name %></li>
  <% end %>
</ul>
```

The `all` statement is going to work fine if you'd like to retrieve a few hundred records, however if you'd like to retrieve thousands of records or more in order to for instance export them to a CSV file or send out an e-mail newsletter then you'll want to use `find_each`, a method created expressly for the purpose of efficiently querying for a large amount of data (more than 1,000 records). To use `find_each` when you'd like to retrieve all of a model's records, just append it to `all`, like this:

```
>> Location.all.find_each do |location|
  ?> puts location.name
>> end
```

If your intent is to *display* a large number of records to the user (as opposed to performing a task such as CSV or PDF generation), then you'll want to *paginate* the query results. You'll learn all about an easy way to paginate records in the later section, "Paginating Results".

Create a Locations Listing Page

ArcadeNomad gives users the ability to navigate the locations and click on a location name to learn more about that location and its games. Let's create a simplified version of this feature. ArcadeNomad embraces [RESTful routing](http://edgeguides.rubyonrails.org/routing.html)⁵² throughout, a best practice encouraged by the Rails community. Therefore let's create the `locations` controller and its associated index action, used to display a list of arcade locations:

⁵²<http://edgeguides.rubyonrails.org/routing.html>

```
$ rails g controller locations index
create  app/controllers/locations_controller.rb
       route get "locations/index"
invoke  erb
create  app/views/locations
create  app/views/locations/index.html.erb
<snip>
create  app/assets/stylesheets/locations.css.scss
```

Despite promoting the RESTful routing approach, Rails inexplicably does not update the `config/routes.rb` file to reflect the RESTFUL routing pattern. Open up `config/routes.rb` and locate this line:

```
get "locations/index"
```

Replace it with this line:

```
resources :locations
```

Save the changes, and then open up the newly created `app/controllers/locations_controller.rb` file, where you'll find the `index` action skeleton:

```
def index
end
```

Modify the `index` action to look like this:

```
def index
  @locations = Location.all
end
```

Finally, open the `app/views/locations/index.html.erb` file, delete the placeholder text, and add the following contents:

```
<ul>
  <% @locations.each do |location| %>
    <li><%= location.name %></li>
  <% end %>
</ul>
```

With the changes in place, take a moment to add a few locations to your `locations` table (using the console is your easiest route). If you don't have the `ArcadeNomad` application running, start the Rails server and navigate to `http://0.0.0.0:3000/locations` and you should see a bulleted list of locations, rendered using HTML that looks like this:

```
<ul>
  <li>Buffalo Wild Wings Hilliard</li>
  <li>Plain City Lanes & Pizza</li>
  <li>Galaxy Games and Golf</li>
</ul>
```

While this is a great start, there's plenty of room for improvement. Among other things we could sort the results by name, paginate them for easier navigation, and add more information than just the arcade name. Read on to learn how this is done!

Selecting Specific Columns

For performance reasons you should to construct queries that retrieve the minimal data required to complete the desired task. For instance if you're constructing a list view that only displays the location name and description, there is no reason to retrieve the street address, zip code, and description. You can restrict which columns are selected using the `select` method, as demonstrated here:

```
@locations = Location.select('name, description').order('name')
```

Counting Records

To count the number of available records for a given model, use the `count` method:

```
@count = Location.count
```

In the view you could use Rails' `pluralize` view helper to present a grammatically correct synopsis of the record count:

```
Tracking <%= pluralize(@count, "location") %>
```

The `pluralize` view helper is useful in that it will present the singular or plural version of the second argument based upon the value of the first argument. For instance, if `@count` is 0 then the output will look like this:

```
Tracking 0 locations
```

If `@count` is 1 then the output will look like this:

Tracking 1 location

Finally, if @count is set to a value greater than 1 then the output will look like this:

Tracking 2 locations



When working with associations (introduced in Chapter 4), you likely will *not* want to use the `count` method, because it involves a second database query which is probably unnecessary when you'd like to know for instance how many games are associated with a particular location. See Chapter 4 for more information.

Determining Existence

If your sole goal is to determine whether a particular record exists, *without* needing to actually load the record if it does, use the `exists?` method. For instance to determine whether a manufacturer named `Capcom` exists use the following statement:

```
>> Manufacturer.exists?(name: 'Capcom')
Manufacturer Exists (1.1ms)  SELECT  1 AS one FROM `manufacturers`
WHERE `manufacturers`.`name` = 'Capcom' LIMIT 1
=> true
```

Ordering Records

ArcadeNomad gives users the ability to view arcades according to category by navigating to <http://arcadenomad.com/categories>⁵³. In order to display the categories in alphabetical order you need to order the results using the `order` method:

```
categories = Category.order('name asc')
```

Note how it's not necessary to include the `all` method when using a clause such as `order`; lacking other filters Active Record will just presume you're interested in retrieving all records and in this case ordering them by the `name` attribute. You are however free to make the intent abundantly clear by including the `all` method, as demonstrated here:

⁵³<http://arcadenomad.com/categories>

```
categories = Category.all.order('name asc')
```

Limiting the Number of Returned Records

To limit the number of returned records you can use the `limit` method. For instance to limit the previous query to retrieving only the names of the first five states you'll chain `.limit(5)` to the statement:

```
>> Category.select('name').order('name asc').limit(5)
Category Load (0.6ms)  SELECT name FROM `categories` ORDER BY name asc LIMIT 5
=> #<ActiveRecord::Relation [#<Category id: nil, name: "Airport">,
    #<Category id: nil, name: "Amusement">,
    #<Category id: nil, name: "Amusement Park">,
    #<Category id: nil, name: "Arcade">,
    #<Category id: nil, name: "Bar">]>
```

Most database tutorials couple any discussion of limiting records with an explanation of how to define the query *offset*, or the number of records that should be skipped before beginning to return records. This is accomplished using the `offset` method. For instance if you wanted to retrieve five records but offset the results to begin at the 10th record of the set that would otherwise be returned were you to not have limited the results, you would use `limit` and `offset` in conjunction with one another, like this:

```
>> Category.select('name').order('name asc').limit(2).offset(5)
Category Load (0.6ms)  SELECT `categories`.`name` FROM `categories`
    ORDER BY name asc LIMIT 2 OFFSET 5
=> #<ActiveRecord::Relation [#<Category id: nil, name: "Bowling Alley">,
    #<Category id: nil, name: "Laundromat">]>
```

While you could use the `limit` and `offset` methods and some clever programming to implement record pagination, it is a commonplace problem that has been already solved many times over. In the later section, “Paginating Results”, I’ll introduce the de facto pagination solution used by the majority of the Rails community (and also used in *ArcadeNomad*).

Retrieving the First or Last Record

For various reasons you might be interested in retrieving solely the first or last record. Because `ActiveRecord::Relation` is iterable, one could reasonably conclude it makes sense to use the `0` index offset or the convenience method `first` to retrieve the first record, and `last` to retrieve the last. This means the statements `locations[0]` and `locations.first` in the below example are equivalent:

```
>> locations = Location.order('name asc')
Location Load (0.1ms)  SELECT `locations`.* FROM `locations` ORDER BY name asc
<snip>
>> locations[0]
=> #<Location id: 5, name: "BW3s Dublin", ...>
>> locations.first
=> #<Location id: 5, name: "BW3s Dublin", ...>
```

However, this is a rather inefficient way to retrieve the first value because the initial query will retrieve *all* records. If you're only interested in the first record you can use the `first` method to do so:

```
location = Location.first
```

This is certainly more efficient, however possibly didn't return the desired record because the `first` method's default behavior is to retrieve the record having the lowest primary key value. You can retrieve the first record as determined by the first returned when ordering in ascending fashion by name by modifying the query to look like this:

```
>> location = Location.order('name asc').first
Location Load (0.3ms)  SELECT `locations`.* FROM `locations`
                        ORDER BY name asc LIMIT 1
```

If you'd like to retrieve the last record in a set, the `last` method works in a similar fashion to `first`. Its default behavior is to retrieve the record having the highest primary key value:

```
>> location = Location.last
Location Load (0.2ms)  SELECT `locations`.* FROM `locations`
                        ORDER BY `locations`.`id` DESC LIMIT 1
```

Therefore in order to retrieve the last record as determined by descending alphabetical order, use `order('name desc')` as was used in the earlier example:

```
>> location = Location.order('name desc').last
Location Load (0.3ms)  SELECT `locations`.* FROM `locations`
                        ORDER BY name desc LIMIT 1
```

Retrieving a Random Record

Suppose you wanted to highlight an “Game of the Day” section on the ArcadeNomad home screen. You could do so by randomly retrieving a record from the games table. While ActiveRecord does not offer a convenience method for retrieving a random record, there are a number of easy approaches one could employ to get the job done. One way involves using MySQL's `rand()` function in conjunction with `order`, as described in [the MySQL documentation](https://dev.mysql.com/doc/refman/5.0/en/mathematical-functions.html#function_rand)⁵⁴:

⁵⁴https://dev.mysql.com/doc/refman/5.0/en/mathematical-functions.html#function_rand

```
>> Location.select('id, name').order("RAND()").first
Location Load (0.3ms) SELECT id, name FROM `locations` ORDER BY RAND() LIMIT 1
=> #<Location id: 6, name: "Pizza Works">
>> Location.select('id, name').order("RAND()").first
Location Load (0.3ms) SELECT id, name FROM `locations` ORDER BY RAND() LIMIT 1
=> #<Location id: 5, name: "BW3s Dublin">
```

This works great, so what's the problem? As it happens, `rand()` is a MySQL-specific function, meaning if you attempted to use this example while running ArcadeNomad atop PostgreSQL or SQLite, the query would fail! Of course, such approaches don't present a problem if you don't plan on ever migrating to another database solution, but even so you can avoid any issues that might arise from such unforeseen eventualities by taking advantage of some of the other database-agnostic methods already introduced:

```
>> Location.offset(rand(Location.count)).first
(3.5ms) SELECT COUNT(*) FROM `locations`
Location Load (0.6ms) SELECT `locations`.* FROM `locations`
ORDER BY `locations`.`id` ASC LIMIT 1 OFFSET 2
```

Notice how in this example not one but two queries are actually executed in order to achieve the desired outcome. The first query determines how many records are found in the `locations` table. This value is passed into *Ruby's* `rand()` function, which will use the value as the upper boundary for choosing a random number between zero and this boundary. Therefore if the `locations` table currently contains 16 rows, then `rand()` would return a number between 0 and 16 (both numbers inclusive).

Retrieving Records by Primary Key

Most web applications present data using a series of list and detail views, with the latter presenting information about a specific arcade location, game, or product. To display information about a specific record you'll need to be sure you first retrieve specifically the desired item and no other. The most foolproof way to do so is by querying a table using a record's primary key. This is done using the `find` method, as demonstrated here:


```
>> Location.find(6)
Location Load (0.4ms)  SELECT `locations`.* FROM `locations`
  WHERE `locations`.`id` = 6 LIMIT 1
=> #<Location id: 6, name: "Pizza Works", description: "Best pizza in Hubbard, O\
hio",
created_at: "2014-01-25 04:57:55", updated_at: "2014-01-25 04:57:55",
street: "433 North Main Street", city: "Hubbard", zip: "44425",
latitude: #<BigDecimal:7fc5ee8593d0,'0.411618939E2',18(27)>,
longitude: #<BigDecimal:7fc5ee859358,'-0.80568425E2',18(27)>, state_id: 36,
category_id: nil>
```

Take a moment to closely examine the above query's return value; unlike `all` or the other previous queries in which multiple records are returned, `find` does not return an `ActiveRecord::Relation`. Instead it returns an object of type `Location`:

```
>> Location.find(3).class
=> Location(id: integer, name: string, description: text, created_at: datetime,
  updated_at: datetime, street: string, city: string, zip: string,
  latitude: decimal, longitude: decimal, state_id: integer,
  category_id: integer)
```

The `find` method also accepts an array of primary keys, should you wish to retrieve multiple records:

```
>> Location.select('name').find([3, 5, 9])
Location Load (1.0ms)  SELECT name FROM `locations`
  WHERE `locations`.`id` IN (2, 5, 6)
=> [#<Location id: nil, name: "BW3s Dublin">,
  #<Location id: nil, name: "Pizza Works">,
  #<Location id: nil, name: "Rusty Bucket">]
```

When using `find` in this manner, an array of `Location` objects will be returned, meaning you can iterate over it as desired.

Creating an Arcade Detail Page

Rails applications commonly use the `find` method to retrieve information about a specific record, passing along the record's primary key. For instance, the URL associated with displaying detailed information about the location associated with the primary key 42 will typically look like this: `http://www.arcadenomad.com/locations/42`. Presuming you're using Rails' RESTful routing capabilities as described earlier in this chapter, when a user accesses this URL Rails will know to execute the `Locations` controller's `show` method. Further, it will know that the URL's trailing component will be accessible via the `params` hash's `id` key. This key is often passed along via a list view. Recall from the earlier section "Retrieving All Records" we output a list of all available locations using the following code:

```
<ul>
  <% @locations.each do |location| %>
    <li><%= location.name %></li>
  <% end %>
</ul>
```

Modify this code to use the `link_to` and `location_path` helpers, creating a link to each location's detail page:

```
<ul>
  <% @locations.each do |location| %>
    <li><%= link_to location.name, location_path(location) %></li>
  <% end %>
</ul>
```

When `link_to` and `location_path` are used in this fashion, a link will be created that looks like this:

```
<a href="/locations/12">Plain City Lanes & Pizza</a>
```

When this link is clicked, Rails default RESTful routing understands that the `Locations` controller's `show` action is being requested. The location ID (in this example, 12) is passed along via the `params` hash using the `id` key. You'll want to use the `find` method within the `show` action to retrieve the desired location:

```
def show
  @location = Location.find(params[:id])
end
```

With the `show` action in place you can set about creating the corresponding view (`app/views/locations/show.html.erb`). Just for sake of illustration update `show.html.erb` to display the location's name and description:

```
<h1><%= @location.name %></h1>

<p>
  <%= @location.description %>
</p>
```

Retrieving Records by Condition

Retrieving records by their primary key is really only useful when we're concerned with presenting a detail view for a particular record, or editing or deleting a record. This is because we want to be absolutely certain the correct record is being viewed or manipulated, a certainty that can only typically be achieved using the primary key. However in almost every other query scenario you'll be searching for records according to one or more conditions. For instance you might wish to find all arcades having the zip code 10002, the number of arcade games released in 1982, or the arcade manufacturers having names beginning with the letter T. Fortunately, Active Record is well-suited for constructing these sorts of queries, and in this section I'll introduce the topic in considerable detail.

Creating Conditions Using the where Method

The `where` method offers the most elementary solution for creating a condition-based query. Its capabilities are best illustrated through a series of examples. For starters, suppose you provided users with the ability to search for arcades by zip code. Because the zip code is merely an attribute of the `Location` model, you'll need to create a query that searches the `locations` table for any record identified by the supplied zip code, something that's not possible using the primary key-focused `find` method. Instead, you'll use the `where` method:

```
>> Location.select('name').where('zip = ?', 43026)
Location Load (0.6ms)  SELECT `locations`.`name` FROM `locations`
  WHERE (zip = 43026)
=> #<ActiveRecord::Relation [#<Location id: nil, name: "Buffalo Wild Wings Hilli\
ard">,
#<Location id: nil, name: "Dave & Buster's Hilliard">]>
```

Even if you're not familiar with the syntax, it is probably apparent that the question mark acts as a placeholder for the second argument, which is in this case the zip code. In the section "Better Security Through Parameter Binding" I'll talk about why this is important, but for now just understand that Active Record uses positional parameter binding, meaning each question mark in the `where` method's first argument will be replaced by the method's subsequent arguments in the order in which they appear. For instance suppose `ArcadeNomad` users request a feature that allows them to filter out arcades not having an associated phone number so they can call around to inquire about hosting a birthday party. Logically they would also want to filter on their city, meaning you would need to supply two parameters:

```
>> Location.where('city = ? and telephone != ?', 'New York City', '')
```

Negating a Conditional

There are plenty of reasons you'll want to create queries to retrieve data based on what the records *don't* contain. For instance, you could retrieve a set of arcades that don't have a telephone number set to `NULL` (meaning a telephone number is available):

```
>> Location.where('telephone IS NOT NULL')
```

This works, but is kind of ugly. Rails 4+ users can clean things up a bit using the `where.not` method:

```
>> Location.where.not(telephone: nil)
```

In fact, returning to the earlier example involving using multiple parameters to search for arcades in New York City that are associated with a phone number, you can chain together `where` and `where.not` to produce an arguably more readable query:

```
>> Location.where('city = ?', 'Columbus').where.not(telephone: nil)
```

Better Security Through Parameter Binding

Active Record supports *parameter binding*, a programmatic approach to preventing malicious users from modifying the intent of the query with potentially catastrophic effect. Let's take a moment to consider exactly how much havoc such an attack (known as *SQL injection*) could conceivably wreak were you to instead pass along the user-provided zip code like this:

```
>> Location.where("zip = '#{params[:zip]}'")
```

The `params[:zip]` would be populated with the value of a GET or POST parameter named `zip` passed into the action. This might come from a search form, list view, or any other similar interface. Because you're blindly passing along `params[:zip]` without validating or escaping its contents, a malicious user could modify the intent of the query in order to return *all* records found in the `locations` table at once, including conceivably those you had intended to keep hidden from view. He could do this by setting the `zip` search field to `' or 1`, causing the following query to be executed:

```
SELECT * FROM locations WHERE zip = '' or 1
```

The `or 1` means “or exists”, meaning the query should return any record that currently exists in the table. Because of this careless mistake, your application's data can be owned by a few keystrokes. Fortunately, Rails offers a very easy safeguard to this type of attack, and it can be enabled by signalling to Rails that you'd like the input parameter to be sanitized before passing it into the query. It's done like so:

```
>> Location.where("zip = ?", params[:zip])
```

The question mark serves as a placeholder. Following the query condition string you'll pass along the parameter you'd like to replace the placeholder following sanitization. If you would like to use multiple conditions, then you'll use multiple placeholders:

```
>> Location.where("city = ? and zip = ?", params[:city], params[:zip])
```

As you can see it is incredibly easy to follow this convention; neglect to do so at your own peril!

Fuzzy Searches

There are many instances where you will want to search the database for incomplete information, or “fuzzy strings”. For instance, the user might be in an unfamiliar city and know only the first few digits of the zip code, or might want to know which local arcades carry any version of “Pac Man” (“Pac Man”, “Ms. Pac Man”, “Baby Pac Man”, or otherwise) and therefore only type Pac into the search field. You can execute these searches using the `LIKE` clause like so:

```
>> Game.where("name LIKE ?", "%Pac%")
```

This produces the proper SQL:

```
SELECT `games`.* FROM `games` WHERE (name LIKE '%Pac%')
```

It is probably tempting to try to construct the query with the percentage signs surrounding the placeholder, an approach I can confirm I’ve fallen victim to more than a few times:

```
>> Game.where("name LIKE %?%", "Pac")
```

This will fail because Active Record will wrap the argument in single quotes before inserting it into the query, producing the following SQL:

```
SELECT `games`.* FROM `games` WHERE (name LIKE %'Pac'%)
```

Dynamic Finders

Active Record’s *dynamic finders* (also known as *magic finders*) were unquestionably the feature that first drew me to Rails a few years ago, offering a syntax that I found so irresistibly readable it was impossible to not learn more about the framework. So what’s a dynamic finder? It’s a way to name your finders in such a way that the desired attribute filter becomes part of the finder name:

```
games = Game.find_all_by_release_date(1982)
```

You could even combine multiple attributes together:

```
games = Game.find_all_by_release_date_and_manufacturer(1982, 'Capcom')
```

While an undeniably cool feature, method names such as this can quickly become unwieldy and altogether unreadable, prompting the Rails team to deprecate this particular approach in Rails 4.0 (and remove it altogether in Rails 4.1), urging developers to instead use this approach:

```
games = Game.where(release_date: 1982, manufacturer: 'Capcom')
```

A more simplistic dynamic finder does remain in the core though. If you're only interested in retrieving *one* record, you can use the `find_by` method:

```
game = Game.find_by(name: 'Space Invaders')
```

Executing SQL

Chances are Active Record is going to meet your needs for every conceivable type of query, however on rare occasion you're going to run into a situation where either the associations aren't configured properly (more on associations in the next chapter) or you simply can't figure out how to properly structure the Active Record statement but need to implement some feature immediately. In such cases you can execute raw SQL using the `find_by_sql` method:

```
>> games = Game.find_by_sql("SELECT g.name, m.name  
                             from games g, manufacturers m WHERE g.manufacturer_id = m.id")
```

Beware that `find_by_sql` may be useful in a pinch, however if you're not careful it can really burn you. I specifically chose this example SQL because it introduces a nasty side effect that isn't so obvious without actually executing the example. I'll retrieve the first element of the returned array, see if you can determine the problem:

```
>> games.first  
=> #<Game id: nil, name: "Amstar Electronics / Centuri">
```

The game name is missing! This is because `find_by_sql` will blindly attach any returned attributes to the object, and because there are two attributes named `name`, the second overwrote the first. You can use an alias to resolve this issue:

```
>> games = Game.find_by_sql("SELECT g.name, m.name as `manufacturer_name`
                             from games g, manufacturers m WHERE g.manufacturer_id = m.id")
>> games.first.manufacturer_name
=> "Amstar Electronics / Centuri">
```

Frankly, we're just getting started in terms of the havoc `find_by_sql` can wreak. I'd go so far to say that if you find yourself repeatedly using this method to retrieve data then either your models aren't structured correctly or you're not taking full advantage of everything that Active Record has to offer. My advice would be to avoid `find_by_sql` whenever possible.

Grouping Records

Grouping records according to a shared attribute provides opportunities to view data in interesting ways, particularly when grouping is performed in conjunction with an aggregate SQL function such as `count()` or `sum()`. Active Record offers a method called `group` that facilitates this sort of query. Suppose you wanted to retrieve a list of arcade game release years and the count of arcade games associated with each year. You could construct the query in MySQL like so:

```
mysql> select release_date, count(name) as `count` from games
       -> group by release_date order by `count` desc;
+-----+-----+
| release_date | count |
+-----+-----+
| 1982         | 24    |
| 1987         | 14    |
| 1980         | 14    |
| 1983         | 14    |
| 1984         | 10    |
| 1988         | 10    |
| 1981         | 10    |
| 1986         | 7     |
| 1985         | 6     |
| 1989         | 6     |
| 1979         | 3     |
| 1976         | 2     |
| 1978         | 2     |
| 1972         | 1     |
+-----+-----+
14 rows in set (0.00 sec)
```

This query can be reproduced in Active Record like so:

```
>> games = Game.select('release_date, count(name) as count').
?>group('release_date').order('count desc')
Game Load (0.5ms)  SELECT release_date, count(name) as count FROM `games`
GROUP BY release_date ORDER BY count desc
```

You can then iterate over the `release_date` and `count` attributes as you would any other:

```
>> games.each do |game|
?> puts "#{game.release_date} - #{game.count}"
>> end
1982 - 24
1987 - 14
1980 - 14
1983 - 14
1984 - 10
1988 - 10
1981 - 10
1986 - 7
1985 - 6
1989 - 6
1979 - 3
1976 - 2
1978 - 2
1972 - 1
```

Filtering Grouped Records With the Having Clause

You'll often want to group records in conjunction with a filter. For instance, what if you only wanted to retrieve a grouped count of arcade games released by year in the years *prior* to 1985? You could use `group` in conjunction with `where`:

```
>> games = Game.select('release_date, count(name) as count').
?> group('release_date').where("release_date < 1985").order('count desc')
Game Load (1.1ms)  SELECT release_date, count(name) as count FROM `games`
WHERE (release_date < 1985) GROUP BY release_date ORDER BY count desc
```

This works because you're filtering on the non-aggregated field. What if you wanted to instead retrieve the same information, but only those years associated with more than 10 arcade games? At first blush it would seem you could use `group` in conjunction with `where` to filter the results:


```
>> games = Game.select('release_date, count(name) as count').
?> group('release_date').where("count > 10").order('count desc')
Game Load (0.6ms)  SELECT release_date, count(name) as count FROM `games`
  WHERE (count > 10) GROUP BY release_date ORDER BY count desc
Mysql2::Error: Unknown column 'count' in 'where clause': SELECT release_date,
count(name) as count FROM `games` WHERE (count > 10) GROUP BY release_date
ORDER BY count desc
```

So what happened? As it turns out you can't use `where` to filter anything calculated by an aggregate function, because `where` applies the defined condition *before* any results are calculated, meaning it doesn't know anything about the `count` alias at the time it attempts to perform the filter. Instead, when you desire to filter on the aggregated result, you'll use `having`. Let's revise the previous broken example to use `having` instead of `where`:

```
>> games = Game.select('release_date, count(name) as count').
?> group('release_date').having("count > 10").order('count desc')
Game Load (0.2ms)  SELECT release_date, count(name) as count FROM `games`
  GROUP BY release_date HAVING count > 10 ORDER BY count desc
>> games.each do |game|
?>  puts "#{game.release_date} - #{game.count}"
>> end
1982 - 24
1987 - 14
1980 - 14
1983 - 14
```

Perfect!

Paginating Results

If you only plan on tracking a few dozen arcades or games then retrieving them using the `all` method and dumping them into an bulleted list view is going to do the job nicely. However we ArcadeNomad developers aren't willing to stop until we've identified the location of every single *Space Invaders*, *Asteroids* and *Rampage* cabinet on the planet, hopefully culminating in the aggregation of thousands of listings. It wouldn't be practical to list all of these locations in a single screen, and so you'll want to *paginate* them.

As I mentioned earlier in this chapter, pagination is accomplished using a series of database queries involving `limit` and `offset` clauses. For instance, to retrieve arcade locations in batches of 10 sorted by name you would execute the following queries (MySQL, PostgreSQL and SQLite):

```
SELECT id, name FROM locations ORDER BY name ASC LIMIT 10 OFFSET 0  
SELECT id, name FROM locations ORDER BY name ASC LIMIT 10 OFFSET 10
```

MySQL, PostgreSQL and SQLite all use a 0-based index, meaning executing the first query is the same as executing:

```
SELECT id, name FROM locations ORDER BY name ASC LIMIT 10
```

Therefore when creating a pagination solution you would need to keep track of the current offset and limit values, the latter of which might be variable if you gave users the opportunity to adjust the number of items presented per page. Further, you would also need to create a user interface for allowing the user to navigate from one page to the next. Fortunately, you can forego implementing any of this and instead take advantage of a fantastic gem called `will_paginate`⁵⁵, created by Mislav Marohnic. `will_paginate` effectively handles all of the features required to implement pagination, extending your models with an easy `paginate` method that accepts limit and offset input parameters, automating the passage of the limit and offset values from one page to the next, and even providing a drop-in pagination interface such as that presented in the following screenshot.

⁵⁵https://github.com/mislav/will_paginate/wiki

Arcades

Buffalo Wild Wings Hilliard

Chuck E. Cheese's Dublin

Dave & Buster's Hilliard

Ethyl & Tank

Galaxy Games & Golf

← Previous

1

2

Next →

Paginating a list of arcades



I'm cheating a bit in this screenshot, because the `will_paginate` gem's pagination UI does not look like this by default. Because `ArcadeNomad` uses Bootstrap 3 for layout styling, the `will_paginate_bootstrap`⁵⁶ gem is also used to override `will_paginate`'s default pagination styling.

Installing `will_paginate`

To install `will_paginate`, add the following line to your `Gemfile`:

```
gem 'will_paginate', '~> 3.0'
```

Save the changes per usual and execute `bundle install` from within your application's root directory:

⁵⁶https://github.com/bootstrap-ruby/will_paginate-bootstrap

```
$ bundle install
...
Using will_paginate (3.0.3)
...
```

With `will_paginate` installed it's time to begin using it within your application!

Integrating `will_paginate`

Integrating the `will_paginate` gem into your application is incredibly easy requiring you to just modify the list view query to use the `paginate` method and integrate the pagination UI into the view. Let's begin by using the `paginate` method to paginate locations:

```
def index

  @locations = Location.paginate(:page => params[:page],
                                :per_page => params[:per_page])

end
```

The `paginate` method accepts two parameters, `:page`, which determines the current page number, and `:per_page`, which determines the position at which records should be retrieved. The page number is important because it's multiplied by the `per_page` value to determine the offset. Keeping in mind that the default offset is 0, `will_paginate` will use the formula $(\text{params}[:\text{page}] - 1) * \text{params}[:\text{per_page}]$ to calculate the offset. Therefore if the user is on the second page, and `params[:per_page]` is set to 30 (the default), then the offset will be $(2 - 1) * 30$, or 30.

Conveniently, after adding this method, you do not have to change any of the iteration logic in the view! The `paginate` method will only restrict which part of the total record set is returned.

With the listing action updated, you'll next want to open the corresponding view and add the `will_paginate` view helper:

```
<%= will_paginate @locations %>
```

You can add the view helper in multiple locations, meaning you could place the paginator both above and below the results if you so desire. Once added, it will render HTML that looks like this:

```

<div class="pagination">
  <span class="previous_page disabled"><- Previous</span>
  <em class="current">1</em>
  <a rel="next" href="/locations?page=2" class="ui-link">2</a>
  <a href="/locations?page=3" class="ui-link">3</a>
  <a href="/locations?page=4" class="ui-link">4</a>
  <a class="next_page ui-link" rel="next" href="/locations?page=2">Next -></a>
</div>

```

This HTML was copied from the first page of paginated output, because as you can see the “Previous” link is disabled and the first page is not linked. The `page=` parameter is automatically appended to the links, which is what is subsequently passed into the `paginate` method. Also, because you’ll almost certainly want to stylize the HTML output, be sure to check out [this list of examples](#)⁵⁷.

In conclusion, there are plenty of useful gems that undoubtedly speed and improve the development process, but I think `will_paginate` ranks right up towards the top in terms of function and practicality!

Creating a Prettier URL with FriendlyId

When linking to a record’s so-called detail page, it’s standard practice to use a path helper such as `location_path(:id)`, which will produce a hyperlink that looks like `http://arcadenomad.com/locations/45`. The record ID (45 in this example) is then retrieved by the controller’s `show` method. While functional, such URLs aren’t particularly friendly to users nor search engines. Google apparently considers the matter of user-friendly URLs to be particularly important, devoting an entire section to improving URL structure in its guide, “[Search Engine Optimization Starter Guide](#)”⁵⁸. A preferable URL for a given location might be `http://arcadenomad.com/locations/16-bit-bar`. Such a URL will not only satisfy search engines’ preference for using words in URLs but also clearly improves readability when users pass along URLs to friends.

But how can you change Rails’ behavior to begin treating a string as the record ID instead of the usual integer-based primary key? Thanks to a fantastic gem called `FriendlyId`, this transformation is fairly painless. Created by [Norman Clarke](#)⁵⁹ and [Adrian Mugnolo](#)⁶⁰, `FriendlyId` gives you the flexibility of using a user-friendly URL by making a `friendly.find` method available for use in finding records using a string while continuing to allow you to find records in the traditional way using the integer-based primary key.

In this section I’ll show you how to configure a model to support the `FriendlyId` gem, giving you the option of integrating this useful capability into your Rails application. But first let’s install the gem.

⁵⁷http://mislav.uniqpath.com/will_paginate/

⁵⁸<http://www.google.com/webmasters/docs/search-engine-optimization-starter-guide.pdf>

⁵⁹<https://github.com/norman>

⁶⁰<https://github.com/symbol>

Installing FriendlyId

If you're using Rails 4, then you'll need the FriendlyId master branch, meaning you can just add the following line to your project Gemfile:

```
gem 'friendly_id'
```

If you're using Rails 3+, you'll instead confusingly need to use the 4.0-stable branch (4.0 refers to the FriendlyId version, not Rails):

```
gem 'friendly_id', :branch => '4.0-stable'
```

After saving the Gemfile changes, run `bundle install` per usual to install the gem. Once installed, run the FriendlyId generator:

```
$ rails generate friendly_id
  create  db/migrate/20140623151805_create_friendly_id_slugs.rb
  create  config/initializers/friendly_id.rb
```

The FriendlyId generator created a new migration and an initialization file. The migration file is only used in conjunction with certain FriendlyId features not discussed in this section, but it doesn't hurt to create the table. To create the table run Rake's `db:migrate` task:

```
$ rake db:migrate
== 20140623151805 CreateFriendlyIdSlugs: migrating =====
-- create_table(:friendly_id_slugs)
   -> 0.0437s
-- add_index(:friendly_id_slugs, :sluggable_id)
   -> 0.0179s
-- add_index(:friendly_id_slugs, [:slug, :sluggable_type])
   -> 0.0139s
-- add_index(:friendly_id_slugs, [:slug, :sluggable_type, :scope], {:unique=>true\
e})
   -> 0.0174s
-- add_index(:friendly_id_slugs, :sluggable_type)
   -> 0.0114s
== 20140623151805 CreateFriendlyIdSlugs: migrated (0.1047s) =====
```

For the remainder of this section I'll presume you're using Rails 4 (and therefore FriendlyId's master branch); if not, please refer to the [FriendlyId README](https://github.com/norman/friendly_id)⁶¹ because there are important incompatibility issues between the Rails 4 and Rails 3 versions.

⁶¹https://github.com/norman/friendly_id

Making the Location Model Sluggable

Let's integrate friendly URL capabilities into ArcadeNomad's `Location` model. You'll need to extend the `locations` schema to add a string column named `slug`:

```
$ rails g migration add_slug_to_locations slug:string
```

This migration shortcut will generate a new migration file containing the desired `add_column` statement however you'll nonetheless need to open up the file and modify it to create an index for the newly added `slug` column. Update the migration file to look like this:

```
class AddSlugToLocations < ActiveRecord::Migration
  def change
    add_column :locations, :slug, :string
    add_index :locations, :slug, unique: true
  end
end
```

Save the file and run the migration task again to update the `locations` table:

```
$ rake db:migrate
== 20140623152607 AddSlugToLocations: migrating =====
-- add_column(:locations, :slug, :string)
   -> 9.6747s
-- add_index(:locations, :slug, {:unique=>true})
   -> 0.0122s
== 20140623152607 AddSlugToLocations: migrated (9.6871s) =====
```

Next, you'll need to modify the `Location` model to use `FriendlyId`. Open the `app/models/location.rb` file and add the following two lines:

```
class Location < ActiveRecord::Base

  extend FriendlyId
  friendly_id :name, :use => :slugged

end
```

This tells `FriendlyId` we want to use the *slugged* version of the `Location` model's `name` attribute. So for instance, Magic Mountain Polaris would use the slug `magic-mountain-polaris`.

Because this is an existing model, we'll need some simple way to update all of the `locations` table's `slug` fields. Open your project's Rails console and execute the following command, which will retrieve each record in the database and then immediately save the record, causing `FriendlyId` to convert the name into a slug and save that slug back to the record as part of the save process:

```
>> Location.find_each(&:save)
```

Once complete, review the locations table and you'll see that the `slug` column has been updated:

```
mysql> select id, name, slug from locations;
+----+-----+-----+
| id | name                | slug                |
+----+-----+-----+
|  1 | Ethyl & Tank         | ethyl-tank         |
|  2 | Buffalo Wild Wings Hilliard | buffalo-wild-wings-hilliard |
|  3 | Plain City Lanes & Pizza | plain-city-lanes-pizza |
...
```

We're almost done. You'll next need to modify the `app/view/locations.index.html.erb` file to refer to the `friendly_id` attribute within the location path helper. In earlier examples the default Rails approach was used:

```
<li><%= link_to location.name, location_path(location) %></li>
```

You'll want to modify the helper to look like this:

```
<li><%= link_to location.name, location_path(location.friendly_id) %></li>
```

If you reload the locations list view, you'll see that the location URLs have been modified to look like this:

```
http://arcadenomad.com/locations/plain-city-lanes-pizza
```

Alright! We're now using friendly URLs. Finally, modify the `LocationsController`'s `show` action to use the `friendly` method:

```
@location = Location.friendly.find(params[:id])
```

Save the change and click on one of the locations found in the locations list view. You'll be taken to the appropriate location, but this time via the friendly URL!

Automatically Updating the Slug

Because in the above example the `name` attribute dictates the `slug` value, you might want to automatically update the `slug` column whenever the associated attribute changes. This could be done using a callback however `FriendlyId` offers a convenience method to do just this. Add the following method to your `FriendlyId`-enabled model:


```
def should_generate_new_friendly_id?  
  name_changed?  
end
```

Of course, such conveniences can come with significant consequences. If the search engines have already crawled your website, any FriendlyId-generated URLs will *break* should the slug subsequently change unless you've taken additional steps to configure FriendlyId's [History module](http://rubydoc.info/github/norman/friendly_id/master/FriendlyId/History)⁶².

Closing Considerations

As this section demonstrated, FriendlyId offers a pretty fantastic feature at very little cost in terms of the time required to update your application. There are however a few considerations to keep in mind when using this gem.

For starters, the slug-based approach demonstrated here is just one of several supported by FriendlyId; if your table already happens to use a column named `slug` then you're either going to need to change that column name or use another approach because FriendlyId is unable to accommodate alternative names.

Second, updating slugs is a *big deal* because it could produce 404s since search engines may have already crawled the website and users may have bookmarked certain pages. You'll definitely want to have a look at the aforementioned History module if such changes are foreseen.

Introducing Scopes

Applying conditions to queries gives you the power to retrieve and present filtered data in every imaginable manner. Some of these conditions will be used more than others, and Rails provides a way for you to cleanly package these conditions into easily readable and reusable statements. Consider a filter that only retrieves locations having a website. You could use the following where condition to retrieve these locations:

```
locations = Location.where("url <> ''")
```

You can define a *scope* that will allow you to instead retrieve these locations like this:

```
locations = Location.has_url
```

Scopes are defined using the `scope` class method, passing along the desired scope name in conjunction with a callable object. You'll define scopes within the model in which you'd like to use them. In the below example I'll define the `has_url` scope:

⁶²http://rubydoc.info/github/norman/friendly_id/master/FriendlyId/History

```
class Location < ActiveRecord::Base

  scope :has_url, -> { where("url <> ''") }

end
```

Those developers unfamiliar with Active Record scopes tend to place them on some sort of special pedestal; I did the same when I first encountered this very cool feature. However, consider this: scopes are nothing more than a shortcut for creating class methods! The scope we defined above behaves *exactly* like the following class method:

```
class Location < ActiveRecord::Base

  def self.has_url
    where("url <> ''")
  end

end
```

You can pass arguments into a scope. For instance the following scope will retrieve only those games released in a specific year:

```
class Game < ActiveRecord::Base

  scope :released_in, ->(year) { where('release_date = ?', year) }

end
```

Once defined you can retrieve all games released in 1984 like this:

```
>> games = Game.released_in(1984)
```

Finally, you can chain scopes together to create even more powerful filters. Suppose you wanted to give ArcadeNomad users the option of reviewing only those arcades having both a recorded website URL and phone number. So you create a new Location scope named `has_telephone`. With the new scope in place you can chain both `has_url` and `has_telephone` together like this:

```
locations = Location.has_url.has_telephone
```

In the next chapter we'll talk more about scopes, notably how you can use them in conjunction with associations to create really powerful queries.

Testing Your Scopes

You'll logically want to confirm your custom scopes are working as desired. So how might we go about testing them using RSpec? Let's consider a few examples, starting with confirming no games are returned when searching for games having a release date not found in the test database. Suppose this is what the `Game` factory looks like:

```
FactoryGirl.define do
  factory :game do
    name 'Space Invaders'
    release_date '1978'
    description 'Space Invaders was one of the early blockbuster shooting games.'
  end
end
```

Within the `games_spec.rb` file we use `before(:each)` to setup the factory object:

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

In `before(:each)` we're *building* rather than *creating* the `Game` object, meaning it's not yet in the database when each test runs. So we'll first want to save the object, and then use the `released_in` scope to determine whether any games associated with a known non-existent release date exist in the database:

```
it 'returns no games when searching for non-existent release date' do

  @game.save

  expect(Game.released_in(2044).count).to eq 0

end
```

Using the same factory and `before(:each)` block, we can confirm the `released_in` scope returns a game associated with a known existent release date:

```
it 'returns one game when searching for an existing release date' do

  @game.save

  expect(Game.released_in(@game.release_date).count).to eq 1

end
```

Conclusion

If you've made it this far, congratulations! You've completed what I'd qualify as the beginning of this book and are ready to make the move into the intermediate levels. It's going to be a bumpy ride at times but the trip will be well worth it!

Chapter 4. Introducing Associations

Thus far we've been taking a fairly simplistic view of the ArcadeNomad database, interacting with the tables as if each one were an island. However, in the real world an application's database tables are like an interconnected archipelago, with bridges connecting two or even more islands together. These allegorical connectors make it possible to perform tasks such as determining which locations offer "Space Invaders", ensuring that all locations found in the state of Ohio always use the two letter OH abbreviation for the state, and identifying locations in close proximity to the user's current position. Such relationships are possible thanks to a process known as *database normalization*⁶³, an approach to data organization that eliminates redundancy, formally structures relations, and improves maintainability. Active Record works in conjunction with a normalized database to provide a host of features useful for building and traversing relations with incredible dexterity. In this chapter I'll introduce you to these wonderful capabilities!

Normalizing Your Database

A *relational database* is so-named because it facilitates the creation and management of data according to the rules defined by the *relational model*⁶⁴. The data found in a relational database is conceptually organized into a series of *tables* consisting of *rows* and *columns*. This data can be interrelated using a key-based system in which *primary keys* uniquely identify each row in a table. These primary keys can be referenced as *foreign keys* within other tables in order to codify the relationship between two rows residing in separate tables. For instance, the ArcadeNomad database includes a table named `locations` for storing information about the places housing arcade games. We want each row in this `locations` table to be uniquely identifiable, and so include an automatically incrementing integer-based column named `id` in the table definition. By referencing each location using this `id`, we can always be certain we're viewing or editing the desired location.

The `locations` table includes several fields used to manage the location address, including `street`, `city`, `zip`, and `state_id`. All of these field names are probably self-explanatory, except for the last (`state_id`). This field is so-named because it contains an integer-based foreign key pointing to a row in the `states` table, which contains the names and two-letter abbreviations of the U.S. states (Florida, Ohio, Texas, etc.). While it's possible to just manage the state's name in a `varchar` field of the `locations` table, it is preferable to *constrain* the available values using a predefined set managed within a separate table, and then refer to the states' primary key ID when associating a location with a state. In doing so, we eliminate the possibility of users mistyping "Ohio" as "Oho" or "OhiO", a mistake that would make it difficult or altogether impossible to among other things

⁶³http://en.wikipedia.org/wiki/Database_normalization

⁶⁴http://en.wikipedia.org/wiki/Relational_model

group locations according to state. Further, by managing the states separately, we have the luxury of managing more information about a state than simply its name, including for instance the two-letter abbreviation and perhaps even the state motto, useful attributes which would otherwise not be possible to incorporate into the application were we to instead just store the name in the `locations` table using a free-form fashion.

The ideas and process behind normalizing a database are formalized in a series of what are known as *normal forms*⁶⁵. If you're not familiar with normal forms and have yet to create a normalized database in a past project, I suggest taking some time to research the matter before proceeding with this chapter, as much of what is discussed will make much more sense if you have at least a rudimentary understanding of normalization fundamentals.

Introducing Associations

Although relatively simplistic as compared to many other applications, ArcadeNomad's database very much resembles the allegorical archipelago mentioned at the beginning of this chapter. For instance, each location found in the `locations` table is mapped to a single U.S. state (U.S. states are defined in the `states` table) and category (categories are defined in the `categories` table). Each game (stored in the `games` table) is tied to a manufacturer (defined in the `manufacturers` table). And of course each location is associated with one or more arcade games; conversely we could also say each arcade game is associated with one or more locations. So how are these relations formally defined in a Rails application? Furthermore, how does one go about traversing a relation to for instance know specifically which games are associated with a given location? Thanks to an Active Record feature known as *associations*, such tasks are surprisingly easy once you have the hang of things.

Rails supports several types of associations, and in this chapter I'll introduce you to the most popular, including `belongs_to`, `has_one`, `has_many`, `has_and_belongs_to_many`, and `has_many :through`. Let's kick things off with a brief introduction to each type of association:

- The `belongs_to` Association: The `belongs_to` association is perhaps the easiest of the bunch to understand. It defines a relationship in which each record represented by the model responsible for *defining* the association refers to one record represented by another model. For instance each location is mapped to the U.S. state in which it resides, therefore each location "belongs to" a state.
- The `has_one` Association: Like `belongs_to`, the `has_one` association defines a one-to-one relationship between two models. It is however different from `belongs_to` because the model declaring a `belongs_to` association is also responsible for housing the foreign key, whereas the model declaring a `has_one` association expects the foreign key to reside in the *other* model. It's admittedly difficult for me to imagine a `has_one` example in the context of ArcadeNomad, so let's dream for a moment. Suppose ArcadeNomad becomes a smashing success, and with it comes employees, departments, and managers. Each department is supervised by one

⁶⁵http://en.wikipedia.org/wiki/Database_normalization#Normal_forms

manager, therefore a department has_one manager. In turn, the Manager model would contain the foreign key department_id, and would belong_to the department.

- The has_many Association: A has_many association is defined when a record represented by the model responsible for defining the association can be related to *more than one* record represented by another model. For instance each U.S. state can host more than one location, therefore each U.S. state *has many* locations.
- The has_and_belongs_to_many Association: Each location can host multiple games, and each game can be found within multiple locations. This means that a game “has and belongs to many” locations, and an location “has and belongs to many” games.
- The has_many :through Association: The has_and_belongs_to_many scenario defined above is suffice if you don’t need to do *anything* but maintain a relationship of the described nature between two models. If you want to attach other attributes to that relationship, validate the relationship attributes, or attach callbacks to the relationship, you’ll want to instead use the has_many :through association. The most fundamental difference between has_and_belongs_to_many and has_many :through is that the former is managed solely through a *join table* (more on this later), whereas the latter is managed via a model defined expressly for the purpose of managing the relation.

Let’s spend some additional time digging into each type of association, accompanied by multiple real-world examples and implementations.



A sixth type of association also exists called has_one :through. However I find it to be so infrequently used that I’ve decided not to cover it in this first edition. In a following update I’ll be sure to add a section on the topic.

Introducing the belongs_to Association

You’ll define a belongs_to relationship when each record represented by a particular model refers to a record represented by another model via a foreign key. For instance, the Location model represents the locations found throughout the United States, and defines key location attributes such as the name, street address, city, and *state*. Because the list of available U.S. states are managed in a separate table (states), each record found in the locations table refers to its corresponding state by identifying a foreign key found in the states table. For instance, “Ethyl & Tank” is located in Columbus, Ohio. If you were to peer into the database, this location’s record would look something like this:

```
+-----+-----+-----+-----+-----+
| id | name          | street          | city          | zip   | state_id |
+-----+-----+-----+-----+-----+
| 3  | Ethyl & Tank  | 19 13 Avenue   | Columbus      | 43201 | 36       |
+-----+-----+-----+-----+-----+
```

Note how this record includes a column named `state_id`, and that value is 36. If you were to have a look at the `states` table, notably the record associated with the primary key ID of 36, you'll see that the record pertains to the state of Ohio:

```
mysql> select id, name, abbreviation from states where id = 36;
```

```
+-----+-----+-----+
| id | name | abbreviation |
+-----+-----+-----+
| 36 | Ohio | OH           |
+-----+-----+-----+
```

You'll define a `belongs_to` association *in the model that references the foreign key*. Repeat this several times because many beginners misunderstand which side of the relation should define the association, leading to chaos down the road. Let's work through the process of recreating the `Location` and `State` models, and defining the `belongs_to` relation between the two. Begin by creating the `State` model:

```
$ rails g model State name:string abbreviation:string{2}
```

After running the migration the database will contain a `states` table that looks like this:

```
mysql> describe states;
```

```
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id             | int(11)       | NO   | PRI | NULL    | auto_increment |
| name           | varchar(255)  | YES  |     | NULL    |                |
| abbreviation    | varchar(2)    | YES  |     | NULL    |                |
| created_at     | datetime      | YES  |     | NULL    |                |
| updated_at     | datetime      | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Next let's create the `Location` model:


```
$ rails g model Location name:string street:string city:string state:references
```

This command isn't really any different from the generator used to create the State model, except for the `state:references` shortcut. This tells the generator we want the Location model to *reference* the state model. This shortcut will add the following statement to the newly created model:

```
class Location < ActiveRecord::Base

  belongs_to :state

end
```

Run the migration, and after it's complete you'll find the following locations table in the database:

```
mysql> describe locations;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
name	varchar(255)	YES	MUL	NULL	
street	varchar(255)	YES		NULL	
city	varchar(255)	YES		NULL	
state_id	int(11)	YES	MUL	NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	

With the models, tables and association defined, we can start connecting location and state data together and subsequently traversing the association!

When creating a new location you'll want to associate a state along with assigning the other attributes. There are plenty of ways to go about this, which is in my opinion a detriment because the variety of options can be confusing to beginners. Even so, I'm going to demonstrate several different approaches so you have a well-rounded understanding of what's available. Let's start with what is perhaps the most obvious approach in which a state is created and then its id is assigned to a location object's `state_id` attribute. Let's begin by creating the state:

```
>> state = State.create(name: "Ohio", abbreviation: "OH")
=> #<State id: 36, name: "Ohio", abbreviation: "OH",
    created_at: "2014-06-26 15:55:26",
    updated_at: "2014-06-26 15:55:26">
```

Obviously you're not required to create the parent record prior to performing the assignment; if the record already exists you can retrieve it using `find` or a similar method.

Next we'll create the location:

```
>> location = Location.new(name: "Rubino's Pizza",
    street: "2643 E. Main Street", city: "Bexley")
```

The newly created location's `state_id` is at this point `nil`, because we've yet to assign a value. Let's perform the assignment now:

```
>> location.state_id = state.id
=> 36
```

Finally, let's save the location:

```
>> location.save
SQL (0.7ms) INSERT INTO `locations` (`city`, `created_at`, `name`,
`state_id`, `street`, `updated_at`) VALUES ('Bexley', '2014-06-26 16:38:36',
'Rubinos', 36, '2643 E. Main Street', '2014-06-26 16:38:36')
```

With the association in place, it is incredibly easy to reference a location's associated state by name, as demonstrated here:

```
>> location.state.name
=> "Ohio"
```

Of course, you're free to reference any attribute attached to the `State` model, such as the `abbreviation`:

```
>> location.state.abbreviation
=> "OH"
```

Returning to the matter of forming the association, while using the `id` directly gets the job done, it isn't a particularly graceful solution. After all, the convenient object-oriented interface is one of the reasons to use an ORM such as Active Record in the first place. Indeed, Active Record offers a much more attractive solution, in fact several, and I think you'll agree these are preferable to the previously demonstrated approach.

With the association in place, Active Record understands that the `Location` model's `state_id` attribute will reference a record of type `State`, and therefore allow you to assign the association like so:

```
>> state = State.create(name: "Ohio", abbreviation: "OH")
>> location = Location.new(name: "Rubino's Pizza",
?> street: "2643 E. Main Street", city: "Bexley")
>> location.state = state
```

Now that's certainly an improvement! There are however two other approaches that could cut down on a bit of code. The first will return a new parent object and create the association, but not save the parent object until the child object's save method is called:

```
>> location = Location.new(name: "Rubino's Pizza",
?> street: "2643 E. Main Street", city: "Bexley")
>> location.build_state(name: "Ohio", abbreviation: "OH")
=> #<State id: nil, name: "Ohio", abbreviation: "OH", created_at: nil,
    updated_at: nil>
>> location.save
(0.1ms) BEGIN
SQL (0.2ms) INSERT INTO `states` (`abbreviation`, `created_at`, `name`,
    `updated_at`) VALUES ('OH', '2014-06-26 18:18:06', 'Ohio',
    '2014-06-26 18:18:06')
SQL (0.2ms) INSERT INTO `locations` (`city`, `created_at`, `name`, `state_id`\
,
    `street`, `updated_at`) VALUES ('Bexley', '2014-06-26 18:18:06',
    'Rubino's Pizza', 36, '2643 E. Main Street', '2014-06-26 18:18:06')
(1.3ms) COMMIT
```

Because we *built* the object, it will not be saved until the child object is saved. Therefore, as you can see in the above console output, when `location.save` is called Active Record will first insert the new state into the `states` table, and then insert the new location, referencing the newly created states record ID in the process.

You have the option to *create* the parent immediately using the `create_state` method, as demonstrated here:

```
>> l = Location.new(name: "Rubino's Pizza", street: "2643 E. Main Street",
?> city: "Bexley")
>> l.create_state(name: "Bazinga", abbreviation: "BZ")
(0.1ms) BEGIN
SQL (0.5ms) INSERT INTO `states` (`abbreviation`, `created_at`, `name`,
    `updated_at`) VALUES ('BZ', '2014-06-26 18:26:54', 'Bazinga',
    '2014-06-26 18:26:54')
(1.6ms) COMMIT
>> l.save
```

```
(0.1ms) BEGIN
SQL (0.2ms) INSERT INTO `locations` (`city`, `created_at`, `name`, `state_id`\
,
`street`, `updated_at`) VALUES ('Bexley', '2014-06-26 18:18:06',
'Rubino's Pizza', 36, '2643 E. Main Street', '2014-06-26 18:18:06')
(1.3ms) COMMIT
```

As you can see, once `l.create_state` is executed, the record will immediately be persisted to the database. When `l.save` is called, the newly added state's `id` value will be attached to the location's `state_id` attribute.

So there you have it: several different solutions to the challenge of formalizing the `belongs_to` relationship between two models. So how can you validate this relationship to ensure the association is always performed? That's the topic of the next section.

Validating a `belongs_to` Association

In Chapter 2 you learned how to attach validations to your models to prevent for instance a location being saved to the database without a name or street. Similarly, it would be catastrophic to allow for locations to be saved without an associated state. Fortunately, attaching a validator to a `belongs_to` association is easy:

```
class Location < ActiveRecord::Base

  # Create the association
  belongs_to :state

  # Validate the association
  validates :state, presence: {
    message: 'A state must be assigned to the location'
  }

end
```

With the validation in place, let's fire up the console once again and test it out:

```
>> location = Location.new(name: "Rubino's Pizza", street: "2643 E. Main Street"\  
,  
  ?> city: "Bexley")  
>> location.valid?  
=> false  
>> location.errors.messages  
=> {:state=>["A state must be assigned to the location"]}
```

You might be tempted to instead use the presence validator to test for the existence of `state_id`:

```
validates :state_id, :presence => {  
  message: 'A state must be assigned to the location'}
```

On the surface this makes sense, because if `state_id` is assigned a value then everything is likely fine. However when specifically examining the *attribute* rather than the *relation*, the validator will consider *any* value to be valid:

```
>> location = Location.new(name: "Rubino's Pizza", street: "2643 E. Main Street"\  
,  
  ?> city: "Bexley")  
>> location.state_id = 943  
>> location.valid?  
=> true
```

Wow! That’s certainly not the outcome you might have expected. Therefore, be sure to attach your validator to the association rather than the attribute in order to avoid any unexpected results.

Useful belongs_to Options

The `belongs_to` association supports a number of options that can modify the association’s behavior in useful ways. I’ll introduce the most commonly used options in this section, accompanied by several examples.

Changing the Parent Class Name Rails eliminates much of the time and effort you’d have to spend making mundane decisions about application and database conventions by simply establishing a set of conventions for you. Consider for instance the following `belongs_to` association:

```
belongs_to :state
```

Rails will by default presume the model associated with the parent table is `State`. However, for a variety of reasons the class name might be different, `Estado` for instance (Spanish for “state”). You can override the default class naming convention using the `class_name` option:

```
belongs_to :state, class_name: 'Estado'
```

Changing the Foreign Key Just as Rails makes certain assumptions regarding the name of a `belongs_to` parent model's table name, it also assumes the child model's table will include an attribute named using the parent model's name concatenated with `_id`. (for instance, `state_id`). If you'd like for some reason to override this convention, you can use the `foreign_key` option:

```
belongs_to :state, foreign_key: 'estado_id'
```

Caching Association Counts I can't introduce this particular option without jumping ahead a bit, as it involves using `belongs_to` in conjunction with the `has_many` association which has yet to be introduced. Therefore if this is confusing, return to this section after you've read the rest of the chapter. With that disclaimer out of the way, consider that it is often useful to know how many locations are associated with a particular state. This information could be used when presenting a list of the U.S. states in a list group, akin to how <http://arcadenomad.com/categories>⁶⁶ works by presenting each category name and the number of associated locations. The easiest way to retrieve this count is by formalizing *the other end* of the `belongs_to` relationship by indicating that a state *has_many* locations:

```
class State < ActiveRecord::Base

  has_many :locations

end
```

With this in place, you can easily determine how many locations are associated with a particular state:

```
>> state = State.find_by_abbreviation("OH")
>> state.locations.size
(0.3ms) SELECT COUNT(*) FROM `locations` WHERE `locations`.`state_id` = 36
=> 84
```

However, as you can see by the above example, determining this number requires execution of a `COUNT(*)` query, something we'd like to avoid if possible. Using the `counter_cache` option, you can! When enabled, this option will update a field in the *parent* model's table every time the number of associated records changes, and then whenever `size` is called, Active Record will instead retrieve the value found in that field rather than perform the costly `COUNT(*)` query.

Enabling this option is actually a *two* step process. First you'll want to add the `counter_cache` option to the `belongs_to` declaration:

⁶⁶<http://arcadenomad.com/categories>

```
class Location < ActiveRecord::Base

  belongs_to :state, counter_cache: true

end
```

Then, you'll want to create a migration affecting the parent model's table, adding an integer-based column named `locations_count`:

```
$ rails g migration add_locations_count_to_states locations_count:integer
```

This will generate the following migration:

```
class AddLocationsCountToStates < ActiveRecord::Migration
  def change
    add_column :states, :locations_count, :integer
  end
end
```

Note that the field name (`locations_count`) is adapted to the name of the child model's table. You'll need to modify the name accordingly for your own application's purposes.

Run this migration and Active Record will begin updating the `locations_count` field every time the count changes due to the addition or deletion of a relevant association. But it does *not* take into account the number of associations that might already be in place. Therefore if you add this feature after data has already been loaded into the application, you'll need to take additional steps to ensure the count is correct. This is easily accomplished by adding code similar to the following to `seeds.rb` and executing the `rake db:seed` task within the appropriate environment:

```
states = State.all
states.each do |state|
  state.locations_count = state.locations.count
  state.save
end
```

Updating the Parent Object's Timestamp If you navigate to one of ArcadeNomad's category pages, such as <http://arcadenomad.com/categories/bowling-alley>⁶⁷, you'll notice a message under the category indicating the date in which the category was last updated. This date is retrieved from the `categories` table's `updated_at` field. By default this field will only update when some attribute associated with the record changes, however you can modify this behavior so that the `updated_at` field is also updated whenever a child record is associated/disassociated with the parent, or when any of the parent's record's children are materially changed:

⁶⁷<http://arcadenomad.com/categories/bowling-alley>

```
belongs_to :category, touch: true
```

Combining Options You can combine multiple options if necessary. For instance, `ArcadeNomad`'s `Location` model includes the following `belongs_to` association:

```
belongs_to :category, touch: true, counter_cache: true
```

Using Conditions with a `belongs_to` Association

It's certainly convenient to be able to easily retrieve a list of games associated with a particular location, but what if some `ArcadeNomad` users were game snobs (ahem, aficionados), and they only wanted to see games at a specific location that were manufactured by Atari? You can attach a condition to the query that references a particular attribute in the `Game` model:

```
>> location = Location.find_by(name: '16-bit Bar')
>> manufacturer = Manufacturer.find_by(name: 'Atari')
>> location.games.where('manufacturer_id = ?', manufacturer.id)
```

Using Scopes on a `belongs_to` Association

One of the cool things about scopes (introduced in Chapter 2) is the ability to use them in conjunction with associations. You might recall in the last chapter we created a `Game` scope that enabled games to be easily filtered according to the year in which they were released. You can use this scope to retrieve only those games hosted at 16-bit Bar released in 1984:

```
>> location = Location.find_by(name: '16-bit Bar')
>> location.games.released_in(1984)
```

Testing a `belongs_to` Association

As you've learned, creating and validating a `belongs_to` association is pretty straightforward. However, this is programming and there's always room for error, so let's write a few tests to make sure this association is always configured as we expect it to be. Before doing so, we need to install another gem that offers a number of convenience matchers for confirming associations are configured as desired. It's called [shoulda-matchers](https://github.com/thoughtbot/shoulda-matchers)⁶⁸, and it was created and maintained by the fine folks at [thoughtbot](http://thoughtbot.com/)⁶⁹. Install `shoulda-matchers` by adding the following line to your project `Gemfile` within the test group::

⁶⁸<https://github.com/thoughtbot/shoulda-matchers>

⁶⁹<http://thoughtbot.com/>


```
gem 'shoulda-matchers', require: false
```

Run `bundle install` to install the gem, and you're done!

Also, we'll need to update the `Game` factory to ensure the default object has a valid `Manufacturer` association. You can do this using the `association` method:

```
FactoryGirl.define do
  factory :game do
    name 'Space Invaders'
    release_date '1978'
    description 'Space Invaders was an early shooting game blockbuster.'
    association :manufacturer, strategy: :build
  end
end
```

The `strategy: :build` tells `FactoryGirl` to only build the association rather than immediately save the associated object to the database, which is the default behavior when no strategy is defined. This is important, because otherwise if you happen to create multiple objects using the same factory, you could run into validation issues because `RSpec` will attempt to save multiple manufacturers having the same name. See the `FactoryGirl` documentation for more information about associations and build strategies.

With this association defined, `FactoryGirl` will look for a `Manufacturer` factory, using that object as the `Game` factory's manufacturer, meaning you'll also need to create the `Manufacturer` factory (`spec/factories/manufacturers.rb`):

```
FactoryGirl.define do
  factory :manufacturer do
    name 'Midway'
  end
end
```

With `shoulda-matchers` installed and the factories updated, let's begin by simply confirming the `Game` model's `Manufacturer` `belongs_to` association is in place. As with previous test-related examples the following examples assume you're using a factory and `before(:each)` block:

```
it 'should belong to Manufacturer' do
  expect(@game).to belong_to :manufacturer
end
```

In this example we’re using the shoulda-matchers `belongs_to` matcher to confirm the `Game` model does indeed have a `belongs_to` association with the `Manufacturer` model.

Incidentally, some developers far more knowledgeable than me disagree with the practice of testing for associations in this fashion, arguing that `RSpec`’s role is to test model behavior rather than model structure. While I can certainly see their point, personally I don’t mind knowing for certain that I didn’t mistakenly edit or delete the wrong association when refactoring some code (it certainly has happened to me before), and therefore prefer adding these sorts of tests to the mix.

Introducing the `has_one` Association

Like `belongs_to`, the `has_one` association is used when you want to relate one instance of a model with exactly one instance of another. The difference between the two is that while the model responsible for defining the `belongs_to` association also houses the foreign key, when defining a `has_one` association it’s the model referenced by the `has_one` definition that houses the foreign key. Let’s work through an example to make the difference apparent. Returning to the scenario briefly discussed earlier in this chapter, suppose `ArcadeNomad` becomes a global phenomenon and we’re forced to hire several staff members to expand the online empire. New departments are created for building and managing an online store, handling customer support, and continuing to develop `ArcadeNomad.com`. Each department is overseen by a manager, meaning a department “has one” manager.

```
class Department < ActiveRecord::Base
  has_one :manager
end
```

This means the `Manager` model’s corresponding `managers` table would house an attribute called `department_id`. Based on what you’ve just learned about the `belongs_to` association, what does this imply? That’s right, the `Manager` model belongs to the `Department` model! Here’s the association definition:

```
class Manager < ActiveRecord::Base
  belongs_to :department
end
```

Once defined, creating the association is easy. Let’s run through an example:

```

>> d = Department.new(name: 'Customer Support')
=> #<Department id: nil, name: "Customer Support", created_at: nil,
    updated_at: nil>
>> d.build_manager(name: 'Rob Lowe')
=> #<Manager id: nil, name: "Rob Lowe", department_id: nil, created_at: nil,
    updated_at: nil>
>> d.save
    (0.1ms) begin transaction
SQL (0.8ms) INSERT INTO "departments" ... ["name", "Customer Support"] ...
SQL (0.1ms) INSERT INTO "managers" ... ["department_id", 2],
    ["name", "Rob Lowe"] ...
    (4.1ms) commit transaction
=> true
>> d.manager.name
=> "Rob Lowe"

```

If the `build_` method doesn't make any sense, please refer back to the earlier introduction to the `belongs_to` association, where I introduce the different ways in which two models can be related together.

Validating a `has_one` Association

Validating the presence of a `has_one` association is identical to the process used for validating `belongs_to` associations:

```

class Department < ActiveRecord::Base

  has_one :manager

  # Validate the association
  validates :manager, presence: {
    message: 'Every department must be assigned a manager!'
  }

end

```

Updating a `has_one` Association

Suppose one of the company managers is promoted and is now working on a new secret project, meaning a new manager needs to be assigned to the now leaderless department. How one goes about assigning a manager to the department should by now be obvious:

```
>> d = Department.find_by(name: 'Customer Support')
>> m = Manager.find_by_name('Molly Ringwald')
>> d.manager = m
```

This approach shouldn't really be a surprise. But take a moment to think about what happens when a department manager is replaced:

```
(0.1ms) begin transaction
SQL (1.4ms) UPDATE "managers" SET "department_id" = ?, "updated_at" = ?
  WHERE "managers"."id" = 2 [["department_id", nil], ... "]
SQL (0.1ms) UPDATE "managers" SET "department_id" = ?, "updated_at" = ?
  WHERE "managers"."id" = 1 [["department_id", 2], ... ]
(1.1ms) commit transaction
```

Because a department can only “have one” manager, the manager previously assigned to the department now has his `department_id` attribute set to `nil`. Given the particular situation this is the outcome we want because manager Rob Lowe is now working on special projects rather than running a specific department, however such an outcome may be unexpected in other scenarios.

Deleting a has_one Association

To disassociate a manager from a department, you can set the department's manager attribute to `nil`:

```
>> d = Department.find_by(name: 'Customer Support')
>> d.manager = nil
(0.1ms) begin transaction
SQL (0.6ms) UPDATE "managers" SET "department_id" = ?, "updated_at" = ?
  WHERE "managers"."id" = 2 [["department_id", nil],
  ["updated_at", "2014-07-30 17:52:01.674934"]]
(1.4ms) commit transaction
=> nil
```

If you want to remove a `has_one` association *and* delete the manager altogether, you can use the `destroy` method:

```
>> d = Department.find_by(name: 'Customer Support')
>> d.manager.destroy
(0.1ms) begin transaction
SQL (0.4ms) DELETE FROM "managers" WHERE "managers"."id" = ? [{"id", 1}]
(1.6ms) commit transaction
```

Useful has_one Options

The `has_one` association supports a number of options that can change the association's behavior in very useful ways. Many of these options were already introduced in the earlier `belongs_to` section, and the behavior is so similar I wonder about the utility of redundantly introducing them anew. Refer to the earlier section and see the `has_one` documentation [here](#)⁷⁰ for more details.

Testing a has_one Association

Because the `ArcadeNomad` application doesn't happen to use any `has_one` associations (although I'm currently thinking hard about a plausible example for a future update, e-mail me if you have any ideas), I'll have to continue referring to the tangential office hierarchy example described above. To confirm your `Department` model is properly associated with the `Manager` model, you can use the `shoulda-matchers`' `have_one` matcher, as demonstrated here:

```
# This test would appear in specs/models/department_spec.rb
it 'should have one Manager' do
  expect(@department).to have_one :manager
end
```

This example presumes you have the `shoulda-matchers` gem installed, and have created factories for the hypothetical `Department` and `Manager` models. If this doesn't make any sense, see the earlier section, "Testing a `belongs_to` Association".

Useful Links

As mentioned, there's a lot of confusion among beginners regarding the difference between the `belongs_to` and `has_one` associations. Check out the following links for other perspectives on the matter if it's still not clear:

- [Differences Between has_one and belongs_to in Ruby on Rails](#)⁷¹: Neil Rosentech penned a lucid blog entry highlighting the differences between these two associations.

⁷⁰http://apidock.com/rails/ActiveRecord/Associations/ClassMethods/has_one

⁷¹<http://requiremind.com/differences-between-has-one-and-belongs-to-in-ruby-on-rails/>

Introducing the `has_many` Association

When introducing the `belongs_to` association earlier in this chapter I demonstrated how *ArcadeNomad*’s locations were each associated with a specific U.S. state. The `Location` model declares another `belongs_to` association with the `Category` model used to identify the type of location (such as “Bowling Alley”, “Pool Hall”, or “Restaurant”). This sort of relationship is useful for ensuring each location’s category is properly normalized in a way that eliminates inconsistencies. However, you’ll also often want to consider this relationship from the perspective of the parent. For instance, over at *ArcadeNomad.com* you can view a list of locations associated with any category such as “Bowling Alley”: <http://arcadenomad.com/categories/bowling-alley>⁷². This is possible because the `Category` model declares a `has_many` association:

```
class Category < ActiveRecord::Base

  has_many :locations

end
```

The `Location` model in turn defines its association with the `Category` model as `belongs_to`:

```
class Location < ActiveRecord::Base

  belongs_to :category

end
```

Remember, this means the `Location` model’s companion `locations` table will include a column named `category_id`. Because the `Category` model “has many” instances of the `Location` model, any given record found in the `Category` model’s `categories` table can be related to zero or more `Location` records. Indeed you can even treat the locations associated with a category as an array:

```
>> category = Category.find_by(name: 'Barcade')
>> category.locations
Location Load (0.3ms) SELECT `locations`.* FROM `locations`
  WHERE `locations`.`category_id` = 7
=> []
>> location = Location.find_by(name: '16-bit Bar')
>> category.locations << location
SQL (0.5ms) UPDATE `locations` SET `category_id` = 7,
  `updated_at` = '2014-07-30 18:41:27' WHERE `locations`.`id` = 8
>> category.locations.first.name
=> "16-bit Bar"
```

⁷²<http://arcadenomad.com/categories/bowling-alley>

If the location doesn't yet exist, you can optionally use the `build` and `create` methods as was demonstrated in the earlier section introducing the `belongs_to` association. Here's an example:

```
>> category = Category.find_by(name: 'Barcade')
>> category.locations.create(name: "Gino's Pizza", street: "4244 Main Street",
?>..., zip: 43215)
SQL (1.1ms) INSERT INTO `locations` (`category_id`, `city`, ..., `name`, ...)
VALUES (7, 'Columbus', ..., 'Gino\'s Pizza', ...)
```

Validating a `has_many` Association

After having learned how to validate `belongs_to` and `has_one` associations, I'd imagine you already have a pretty good idea how to go about validating a `has_many` associations: just attach a presence validator to the associated model. For instance, it doesn't make any sense to track a manufacturer that doesn't have any associated games, so a presence validator is attached to games in the `Manufacturer` model:

```
class Manufacturer < ActiveRecord::Base

  # Create the association
  has_many :games

  # Validate the association
  validates :games, presence: {
    message: 'Every manufacturer must be associated with at least one game.'}

end
```

With the validation in place, let's fire up the console once again and test it out:

```
>> manufacturer = Manufacturer.new(name: 'Commodore')
>> manufacturer.valid?
=> false
>> manufacturer.errors.messages
=> {:games=>["Every manufacturer must be associated with at least one game."]}
```

Checking for presence isn't the only characteristic you might want to validate in regards to the `has_many` association. What if you wanted to impose a maximum limit on the number of associations, for instance preventing a manufacturer from being associated with more than five games? Even though doing so would be rather illogical, let's implement it for the sake of example:

```
class Manufacturer < ActiveRecord::Base

  validate :maximum_number_games

  has_many :games

  private

  def maximum_number_games
    errors.add(:games, 'No more than 5 games per manufacturer.') if games.size > \
5
  end

end
```

In this example we're using a custom validation method called `maximum_number_games` that checks the number of games associated with the manufacturer. If more than five, an error message will be attached to the object.

Testing a `has_many` Association

You can use the shoulda-matchers `have_many` matcher to confirm the association is configured as desired:

```
it 'should have many games' do
  expect(@manufacturer).to have_many :games
end
```

Presuming you actually did implement the validation responsible for ensuring no more than five games can be associated with a manufacturer, we can write a test for that as well:

```
it 'should be invalid when adding more than five games' do

  6.times {
    @manufacturer.games << FactoryGirl.build(:game)
  }

  expect(@manufacturer).to_not be_valid

end
```


Navigating a has_many Association

Iterating over the locations associated with a category is similarly easy. For instance ArcadeNomad’s Categories controller’s show view uses code very similar to the following to output the locations associated with a particular category:

```
<% if @category.locations.size > 0 %>
  <ul class="list-group">
    <% @category.locations.each do |location| %>
      <li class="list-group-item">
        <%= link_to location.name, location_path(location.id) %>
      </li>
    <% end %>
  </ul>
<% else %>
  <p>No category locations available.</p>
<% end %>
```

Counting has_many Associations

With the has_many association defined, you can easily determine how many locations belong to a specific category:

```
>> category = Category.find_by(name: 'Bowling Alley')
>> category.locations.size
>> 23
```

Active Record will by default use COUNT(*) when querying the database to determine how many records are related via a has_many association. You can (and should) replace this expensive query with an optimized alternative by using the :counter_cache option in conjunction with the belongs_to association. See the earlier section on this topic titled “Caching Association Counts”.



Active Record guru Josh Susser penned an excellent blog post about the difference between count, length, and size. You can read it [here](http://blog.hasmanythrough.com/2008/2/27/count-length-size)⁷³.

Determining has_many Association Existence

However you might only wish to know whether a category is associated with any locations, without necessarily caring about the count. You can use the exists? method for this purpose:

⁷³<http://blog.hasmanythrough.com/2008/2/27/count-length-size>

```
>> category = Category.find_by(name: 'Bowling Alley')
>> category.locations.exists?
Location Exists (0.8ms) SELECT 1 AS one FROM `locations`
  WHERE `locations`.`category_id` = 4 LIMIT 1
=> true
```

You can also determine whether a specific location is associated with a category. There are a few different ways to go about this, including passing in the location object, location id, or even a location-specific attribute such as the zip code. The following three examples demonstrate these different options, beginning by passing along an object of type Location:

```
>> location = Location.find_by_name('Plain City Lanes & Pizza')
>> category.locations.exists?(location)
Location Exists (0.6ms) SELECT 1 AS one FROM `locations`
  WHERE `locations`.`category_id` = 4 AND `locations`.`id` = 3 LIMIT 1
=> true
```

The same outcome can be achieved by passing along only the Location record's id:

```
>> category.locations.exists?(location.id)
```

Finally, you could even pass along a specific attribute. For instance, you could determine how many locations having the zip code 43064 are associated with the category Bowling Alley:

```
>> category = Category.find_by_name('Bowling Alley')
>> category.locations.exists?(zip: '43064')
Location Exists (0.6ms) SELECT 1 AS one FROM `locations`
  WHERE `locations`.`category_id` = 4 AND `locations`.`zip` = '43064' LIMIT 1
=> true
```

Deleting a has_many Association

You have the option of deleting an association via its parent has_many association. Bear in mind this deletes the *association* by setting the child record's relevant foreign key to nil; it does not delete the actual associated record! You can delete the association using the `delete` method, as demonstrated here:

```
>> category = Category.find_by_name('Bowling Alley')
>> location = Location.find_by_name('Plain City Lanes & Pizza')
>> category.locations.delete(location)
SQL (3.4ms) UPDATE `locations` SET `locations`.`category_id` = NULL
WHERE `locations`.`category_id` = 4 AND `locations`.`id` IN (3)
```

Bear in mind the location child record is now *orphaned* as it pertains to categories, meaning it may not be easily accessible in your application list views until the category is updated or you take additional steps to explicitly display unassigned locations under a special “Uncategorized” moniker. You can however modify this default behavior and actually delete the child record based on how the association dependency is defined. See the next section, “Defining has_many Dependency Behavior” for more details.

Defining Dependency Behavior

Suppose an ArcadeNomad administrator has a falling out with the local skating rink and vowed to never again promote a rink of any sort. Nevermind the fact such an idea would be ludicrous; anybody who grew up in the 80’s knows skating rinks are part of the fabric of any civil society. The administrator returns to the office, and in a huff, concludes he will remove the “Skating Rink” category. So what should happen to the locations categorized as skating rinks? Should they also be deleted? Should their `category_id` attribute be set to `null`? Should an error be triggered or an exception be raised, preventing the irate administrator from altogether taking such an action? Skating rink-related comedy aside, such decisions are ultimately left to you given their significant repercussions, however no matter your decision rest assured Active Record provides you with the ability to automate such matters.

If you’d like for all child records to be *deleted* when the parent is deleted, use the `destroy` option:

```
has_many :locations, dependent: :destroy
```

You might recall from the last chapter that destroying an object differs from deleting an object in that when destroying an object, any relevant callbacks will be executed and other dependency checks performed. When deleting an object, the object’s record will be blindly removed from the table without any regard to external logic. Chances are in most cases you’ll want to set the dependency to `destroy` rather than `delete`, the latter of which is demonstrated here:

```
has_many :locations, dependent: :destroy
```

Given the potential significance of suddenly removing a bunch of dependent records from the database, Active Record also offers the option of raising an exception or triggering an error should the parent be associated with one or more children records. To raise an exception, use the `restrict_with_exception` option:

```
has_many :locations, dependent: :restrict_with_exception
```

To trigger an error, use the `restrict_with_error` option:

```
has_many :locations, dependent: :restrict_with_error
```

With the latter option in place, should you attempt to destroy a category associated with one or more children, an error will be triggered and the attempt will fail:

```
>> category = Category.find_by_name('Bowling Alley')
>> category.destroy
(0.1ms) BEGIN
Location Exists (0.3ms) SELECT 1 AS one FROM `locations`
  WHERE `locations`.`category_id` = 4 LIMIT 1
(0.1ms) END
=> false
>> category.errors
=> #<ActiveModel::Errors:0x007f874b625870 @base=#<Category id: 4,
  name: "Bowling Alley", @messages={:base=>["Cannot delete
  record because dependent locations exist"]}>
```

In the next chapter I'll show you how to incorporate these sorts of protective features into ArcadeNomad's administration console.

Useful has_many Options

The `has_many` association supports a number of options that can change the association's behavior in very useful ways. I'll introduce the most commonly used options in this section, accompanied by several examples.

Autosaving When using the `has_many` association, Rails will automatically save any associated objects that are new records, but will not do so for existing objects. An example will clarify nicely:

```
>> category = Category.find_by(name: 'Restaurant')
>> category.locations
=> [#<Location id: 17, name: "Luigi's Pizza", ...
>> category.first.name = "Gino's Pizza"
>> category.save
(0.1ms) begin transaction
(0.0ms) commit transaction
```

As you can see, despite changing the name of a location associated with the category, when the category was saved, nothing happened. You can ensure the changes are persisted to the database by setting `autosave` to `true`:

```
class Category < ActiveRecord::Base

  has_many :locations, autosave: true

end
```

With `autosave` enabled, let's work through the previous example a second time:

```
>> category = Category.find_by(name: 'Restaurant')
>> category.locations
=> [#<Location id: 17, name: "Luigi's Pizza", ...
>> category.first.name = "Gino's Pizza"
>> category.save
(0.1ms) begin transaction
  SQL (0.4ms) UPDATE `locations` SET `name` = 'Luigi\'s Pizza' WHERE `locations`
`.id` = 17
(0.0ms) commit transaction
```

Changing the Child Class Name Rails eliminates much of the time and effort you'd have to spend making mundane decisions about application and database conventions by establishing a set of conventions for you. Consider for instance the following `has_many` association:

```
has_many :locations
```

Rails will by default presume the model associated with the child table is `Location`. However, for a variety of reasons the class name might be different, `Ubicacion` for instance (Spanish for “location”). You can override the default class naming convention using the `:class_name` option:

```
has_many :locations, class_name: 'Ubicacion'
```

Changing the Foreign Key Just as Rails makes certain assumptions regarding the name of a `has_many` child model's table name, it also assumes the child model's table will include an attribute named using the parent model's name concatenated with `_id`. (for instance, `state_id`). If you'd like for some reason to override this convention, you can use the `:foreign_key` option:

```
has_many :locations, foreign_key: 'categoria_id'
```

Introducing the Has and Belongs to Many (HABTM) Association

In order to help gamers find fun classic arcade games to play in the restaurants, bars, and laundromats around them, *ArcadeNomad* requires an efficient way to relate games to locations. This is accomplished by relating select rows in the `games` table to the row in the `locations` table that represents the location where the related games can be played. This is an example of a *many-to-many* relationship, because a location can be associated with many games, and a game can be associated with many locations.

Active Record refers to this sort of association as `has_and_belongs_to_many`, and is often abbreviated as *HABTM*. A HABTM association is defined in just a few steps. First, you'll create the two models you'd like to associate, for instance `Game` and `Location`. With these models created, you'll next create a *join* table that will connect the related primary keys from the `games` and `locations` tables together:

```
$ rails g migration CreateJoinTableGameLocation game location
```

This is a Rails 4-specific migration shortcut, and it generates the following migration file:

```
class CreateJoinTableGameLocation < ActiveRecord::Migration
  def change
    create_join_table :games, :locations do |t|
      # t.index [:game_id, :location_id]
      # t.index [:location_id, :game_id]
    end
  end
end
```

The index definitions are commented out by default, but it is recommended you do enable indexing of these columns; exactly how they should be indexed is dependent upon the database you're using. For instance if you're using MySQL then you should uncomment both lines because of MySQL's particular query optimization strategy, but it may not be useful to do this for other databases. After sorting out the best option for your database, save and run the migration. The table will be named `games_locations` and look like this:

```
mysql> describe games_locations;
+-----+-----+-----+-----+-----+-----+
| Field          | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| game_id        | int(11)   | NO   | MUL | NULL     |       |
| location_id    | int(11)   | NO   | MUL | NULL     |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

If you're still running Rails 3, you'll need to use a slightly different syntax and do a bit more work:

```
$ rails g migration CreateGamesLocationsTable
```

This will generate the following migration file:

```
class CreateGamesLocationsTable < ActiveRecord::Migration
  def up
  end

  def down
  end
end
```

Modify the migration file to look like this:

```
class CreateGamesLocationsTable < ActiveRecord::Migration
  def change
    create_table :games_locations, :id => false do |t|
      t.references :game, :null => false
      t.references :location, :null => false
    end
    add_index :games_locations, [:game_id, :location_id], :unique => true
    add_index :games_locations, [:location_id, :game_id], :unique => true
  end
end
```

Once saved, run the migration to add the table to the database. With the table added, you can define the associations within the Game and Location models:

```
class Game << ActiveRecord::Base

  has_and_belongs_to_many :locations

end
```

And the Location model:

```
class Location << ActiveRecord::Base

  has_and_belongs_to_many :games

end
```

Whether you're running Rails 3 or Rails 4, the join table name *must* list the names of the two reference tables in alphabetical order! If you'd like to override this default behavior, you'll need to specify the name using the `join_table` option when declaring the `has_and_belongs_to_many` relationship:

```
has_and_belongs_to_many :locations, join_table: 'juegos_ubicaciones'
```

With the association created, you can begin using it!

Creating a HABTM Association

With the association defined, you can begin using it! Let's consider a few examples. To add a game to a location, you can conveniently use Ruby's array syntax:

```
>> game = Game.find_by(name: 'Space Invaders')
=> #<Game id: 24, name: "Space Invaders"...
>> location = Location.find(name: '16-Bit Bar')
=> #<Location id: 8, name: "16-Bit Bar"...
>> location.games << game
(0.2ms) BEGIN
SQL (0.4ms) INSERT INTO `games_locations` (`game_id`, `location_id`)
  VALUES (24, 8)
(0.1ms) COMMIT
```

With the changes committed to the database, you should see the following row added to your database:


```
mysql> select * from games_locations;
```

```
+-----+-----+
| game_id | location_id |
+-----+-----+
|      ... |           ... |
|       24 |           8 |
+-----+-----+
```

Keep in mind the same outcome could be accomplished by assigning the *location* to the *game*:

```
>> game = Game.find_by(name: 'Space Invaders')
=> #<Game id: 24, name: "Space Invaders"...
>> location = Location.find_by(name: '16-Bit Bar')
=> #<Location id: 8, name: "16-Bit Bar"...
>> game.locations << location
(0.2ms) BEGIN
SQL (0.4ms) INSERT INTO `games_locations` (`game_id`, `location_id`)
VALUES (24, 8)
(0.1ms) COMMIT
```

You can also optionally use the same `build` and `create` methods first introduced in the section “Introducing the belongs_to Association”. If you recall from that earlier introduction, `build` will create *but not immediately save* the record, while `create` will both create and save the record. Two examples follow, the first involving the `build` method:

```
>> location = Location.find_by(name: '16-Bit Bar')
=> #<Location id: 8, name: "16-Bit Bar"...
>> location.games.build(name: 'Space Invaders')
=> #<Game id: nil, name: "Space Invaders", created_at: nil, updated_at: nil>
>> l.save
(0.1ms) BEGIN
SQL (0.5ms) INSERT INTO `games` (`created_at`, `name`, `updated_at`)
VALUES ('2014-07-01 17:18:33', 'Space Invaders', '2014-07-01 17:18:33')
SQL (0.2ms) INSERT INTO `games_locations` (`game_id`, `location_id`)
VALUES (24, 8)
(0.3ms) COMMIT
=> true
```

Now we’ll accomplish the same task, but this time using the `create` method. Note how the new record and the association are saved immediately upon execution of `create`:

```
>> location = Location.find_by(name: '16-Bit Bar')
=> #<Location id: 8, name: "16-Bit Bar"...
>> location.games.create(name: 'Space Invaders')
(0.1ms) BEGIN
SQL (0.5ms) INSERT INTO `games` (`created_at`, `name`, `updated_at`)
VALUES ('2014-07-01 17:19:52', 'Space Invaders', '2014-07-01 17:19:52')
SQL (0.3ms) INSERT INTO `games_locations` (`game_id`, `location_id`)
VALUES (24, 8)
(0.8ms) COMMIT
```

Setting a Default Sort Order

When listing a location's games, you might want to always sort the results in ascending alphabetical order. To do so you can set the default ordering within the `has_and_belongs_to_many` definition:

```
class Location << ActiveRecord::Base

  has_and_belongs_to_many :games, -> { order('name asc') }

end
```

Once defined, when retrieving a location, its games collection will automatically be ordered alphabetically, as depicted below:

```
>> location = Location.find_by(name: '16-Bit Bar')
>> location.games.map { |g| g.name }
=> ["Asteroids",
    "Centipede",
    "Defender",
    "Dig Dug",
    "Donkey Kong Junior",
    "Donkey Kong",
    "Frogger"]
```

Validating a HABTM Association

You can validate a HABTM association in a few different ways. For instance, if you wanted to ensure a location is associated with at least one game you could create a custom validator to check the size of the games association:

```
class Location << ActiveRecord::Base

  validate :at_least_one_game

  has_and_belongs_to_many :games

  private

  def at_least_one_game
    errors.add(:games, 'Please associate at least one game.') if games.size == 0
  end

end
```

Testing a HABTM Association

To demonstrate testing of `has_and_belongs_to_many` associations, we'll continue using the `shoulda-matchers` gem and `factories` as first discussed in the earlier section, "Testing a `belongs_to` Association". Let's begin with the basic example, demonstrating confirmation of a properly configured association between the `Location` and `Game` models. The following test would be placed in `location_spec.rb`:

```
it 'should have many games' do
  expect(@location).to have_and_belong_to_many :games
end
```

Another useful test involves confirming the desired default sort ordering, as demonstrated in the earlier section, "Setting a Default Sort Order". This example also offers the opportunity to introduce another `FactoryGirl` feature. So far we've only required one type of factory per model, but what if you required more than one? There are plenty of different ways one can go about creating multiple factories, and I'll demonstrate two of them below starting with a simple approach, and then a more effective solution that could be used for multiple different testing purposes. Here's the straightforward approach:

```

it 'should order a locations games alphabetically' do

  game_1 = FactoryGirl.build(:game, name: 'Pac-Man')

  game_2 = FactoryGirl.build(:game, name: 'Ikari Warriors')

  game_3 = FactoryGirl.build(:game, name: 'Shinobi')

  @location.save

  @location.games << game_1
  @location.games << game_2
  @location.games << game_3

  @location.reload

  expect(@location.games).to eq([game_2, game_1, game_3])

end

```

In this example we build three games, choosing unique names for each. Next, the location (built in the `before(:each)` block). Next, the three games are assigned to the location, and finally we verify the ordering is working properly by reloading the record and confirming the retrieved games are retrieved in the order “Ikari Warriors”, “Pac-Man”, and “Shinobi”.

When working with a larger number of objects you might consider FactoryGirl’s sequencing feature. [Getting Started](http://rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md)⁷⁴ documentation for several examples demonstrating the sequencing feature. Sequencing alleviates the hassle of creating for instance separate names for each game by incrementing a number used in the name each time the factory is called. Using sequencing, here is what the revised Game factory would look like:

```

FactoryGirl.define do

  sequence(:name) { |n| "Game #{n}" }

  factory :game do
    name
    release_date '1978'
    description 'Space Invaders was an early shooting game blockbuster.'
    association :manufacturer, strategy: :build
  end
end

```

⁷⁴http://rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

end

With the sequence defined, every time the `Game` factory is used the name will be updated to read “Game #1”, “Game #2”, “Game #3”, and so on. You can then forgo the decidedly verbose approach to building the games by instead doing this:

```
3.times { @location.games << FactoryGirl.build(:game, name: 'Shinobi') }
```

Incidentally, this isn’t the only available solution for auto-generating attribute values. Check out [Faker](https://github.com/stympy/faker)⁷⁵ and [Forgery](https://github.com/sevenwire/forgery)⁷⁶ for more information about what’s possible.

Navigating a HABTM Association

No matter which approach you take, the association is accessible from both the `Game` and `Location` model. For instance, you can iterate over the games associated with a location, like this:

```
>> location = Location.find_by(name: '16-Bit Bar')
>> location.games.each do |game|
?> puts game.name
>> end
Donkey Kong
Rampage
Frogger
```

Similarly, you can iterate over the locations associated with a game:

```
>> game = Game.find_by(name: 'Space Invaders')
>> game.locations.each do |location|
?> puts location.name
>> end
Ethyl & Tank
Buffalo Wild Wings Hilliard
Plain City Lanes & Pizza
Dave & Buster's Hilliard
```

Chances are you’ll want to manipulate the output by for instance ordering the locations by name or filtering the locations according to a particular condition. You can do so by appending one of the clause-oriented query methods introduced in the last chapter. For instance, we can easily output the associated locations in ascending order by name:

⁷⁵<https://github.com/stympy/faker>

⁷⁶<https://github.com/sevenwire/forgery>

```
>> games.locations.order('name asc').each do |location|
?> puts location.name
>> end
Buffalo Wild Wings Hilliard
Dave & Buster's Hilliard
Ethyl & Tank
Plain City Lanes & Pizza
```

You could also filter the results by a specific attribute, such as the location's zip code:

```
>> games.locations.where('zip = ?', 43026).each do |location|
?> puts location.name
>> end
Buffalo Wild Wings Hilliard
Dave & Buster's Hilliard
```

There may be instances where you'll want to modify the default query behavior to produce customized results. I'll show you how this is accomplished in the section, "Modifying HABTM Query Behavior".

Determining Association Existence

You can determine how many games are associated with a location by retrieving the count:

```
>> location = Location.find_by(name: '16-Bit Bar')
>> location.games.size
=> 26
```

However you might only wish to know whether a location is associated with any games, without caring about the number if so. You can use the `exists?` method for this purpose:

```
>> location = Location.find_by(name: '16-Bit Bar')
>> location.games.exists?
Game Exists (0.6ms) SELECT 1 AS one FROM `games` INNER JOIN `games_locations`
  ON `games`.`id` = `games_locations`.`game_id` WHERE `games_locations`.`locatio\
n_id` = 8 LIMIT 1
=> true
```

You can also determine whether a specific game is associated with a location. There are a few different ways to go about this, including passing in the game object, game id, or even a game-specific attribute such as the name. The following three examples demonstrate these different options, beginning by passing along an object of type `Game`:

```
>> location.games.exists?(game)
Game Exists (0.5ms)  SELECT  1 AS one FROM `games` INNER JOIN `games_locations`
  ON `games`.`id` = `games_locations`.`game_id` WHERE
  `games_locations`.`location_id` = 8 AND `games`.`id` = 24 LIMIT 1
=> true
```

The same outcome can be achieved by passing along only the Game record's id:

```
>> location.games.exists?(game.id)
Game Exists (0.5ms)  SELECT  1 AS one FROM `games` INNER JOIN `games_locations`
  ON `games`.`id` = `games_locations`.`game_id` WHERE
  `games_locations`.`location_id` = 8 AND `games`.`id` = 24 LIMIT 1
=> true
```

Finally, you could even pass along a specific attribute. For instance, you could determine how many locations having the zip code 43215 are associated with Space Invaders:

```
>> game = Game.find_by(name: 'Space Invaders')
>> game.locations.exists?(zip: '43215')
Location Exists (0.6ms)  SELECT  1 AS one FROM `locations` INNER JOIN
  `games_locations` ON `locations`.`id` = `games_locations`.`location_id`
  WHERE `games_locations`.`game_id` = 24 AND `locations`.`zip` = '43215' LIMIT 1
=> true
```

Ensuring Unique HABTM Associations

By default there is nothing to stop you from associating the same game to a location multiple times:

```
>> location = Location.find_by(name: '16-Bit Bar')
>> game = Game.find_by(name: 'Space Invaders')
>> location.games << game
(0.2ms)  BEGIN
SQL (0.4ms)  INSERT INTO `games_locations` (`game_id`, `location_id`)
  VALUES (24, 8)
(0.1ms)  COMMIT
>> location.games << game
(0.2ms)  BEGIN
SQL (0.4ms)  INSERT INTO `games_locations` (`game_id`, `location_id`)
  VALUES (24, 8)
(0.1ms)  COMMIT
```

This is almost certainly undesirable behavior. You could use conditional existence checks to ensure a game isn't already associated with the location, like this:

```
>> location.games << game unless location.games.exists?(game)
```

While a plausible solution, I'd rather not want to be bothered with having to perform this sort of check every time I wanted to create a new HABTM association. You'll often see suggestions regarding passing along the `uniq` method (`uniq` is available as an option for Rails 3.X):

```
class Location << ActiveRecord::Base

  has_and_belongs_to_many :games, -> { uniq }

end
```

However, this does *not* fix the problem! It only ensures that distinct (unique) rows are retrieved from the join table:

```
>> location = Location.find_by(name: '16-Bit Bar')
=> #<Location id: 8, name: "16-Bit Bar"...
>> location.games
Game Load (1.7ms)  SELECT DISTINCT `games`.* FROM `games`
  INNER JOIN `games_locations` ON `games`.`id` = `games_locations`.`game_id`
 WHERE `games_locations`.`location_id` = 8
```

Take a close look at the generated query. It uses `DISTINCT` to ensure only unique rows are being retrieved. If you look back at previous examples, you'll see the `DISTINCT` keyword is not included in the queries. Even so, using methods in this fashion is quite useful, and I'll talk about the practice in the section "Modifying HABTM Query Behavior". To prevent duplicate entries though, you'll want to employ a different strategy. Several are available, however the easiest in my opinion is to enforce integrity at the database level by ensuring the `games_location` index is unique. You can do this at the time the table is created by modifying the migration to declare the indexes as unique:

```
class CreateJoinTableGameLocation < ActiveRecord::Migration
  def change
    create_join_table :games, :locations do |t|
      t.index [:game_id, :location_id], :unique => true
      t.index [:location_id, :game_id], :unique => true
    end
  end
end
```

If you've already created and populated the table, no worries because you can always drop and recreate the index using a migration:


```
$ rails g migration addUniqueIndexConstraintToGamesLocationsTable
  invoke  active_record
  create  db/migrate/20140701152223_add_unique_index_constraint_to_games_
         locations_table.rb
```

Open up the newly created migration and modify it to look like this (MySQL-specific; your application indexing strategy for this table might vary slightly):

```
class AddUniqueIndexConstraintToGamesLocationsTable < ActiveRecord::Migration
  def change
    remove_index :games_locations, column: [:game_id, :location_id]
    remove_index :games_locations, column: [:location_id, :game_id]
    add_index :games_locations, [:game_id, :location_id], :unique => true
    add_index :games_locations, [:location_id, :game_id], :unique => true
  end
end
```

Run the migration to complete the process. Note this migration will fail if duplicate records already exist in the `games_locations` table, therefore you may need to perform a bit of cleanup before executing the migration. With the unique constraint in place, any subsequent attempts to insert a duplicate record will raise an `ActiveRecord::RecordNotUnique` exception.

Deleting a HABTM Association

Arcades may occasionally sell or remove a game. To delete a game you can use the `delete` method. You can pass in the object or the object ID to delete an association. Here's an example in which the object is passed into `delete`:

```
>> location = Location.find_by(name: '16-Bit Bar')
=> #<Game id: 24, name: "Space Invaders"...
=> #<Location id: 8, name: "16-Bit Bar"...
>> location.games.delete(game)
(0.2ms) BEGIN
SQL (2.7ms) DELETE FROM `games_locations`
  WHERE `games_locations`.`location_id` = 8 AND `games_locations`.`game_id` = 24
(0.1ms) COMMIT
```

Modifying HABTM Query Behavior

Rails 4 added the ability to modify the default behavior of a `has_and_belongs_to_many` query using a *scope block*. For instance, you can tell Active Record to automatically sort a location's games in ascending fashion according to each game's name:

```
has_and_belongs_to_many :games, -> { order('name ASC') }
```

With the `order` method in place, Active Record will begin modifying the `SELECT` query whenever a location's games are retrieved, adding the `ORDER BY name asc` clause:

```
>> location = Location.find(1)
=> #<Location id: 1, name: "16-Bit Bar"...
>> location.games
Game Load (0.4ms)  SELECT `games`.* FROM `games` INNER JOIN `games_locations`
ON `games`.`id` = `games_locations`.`game_id`
WHERE `games_locations`.`location_id` = 1 ORDER BY name asc
```

Multiple clauses can be chained together within the scope block. For instance, an `approved` attribute might be used to determine whether an `ArcadeNomad` has reviewed and approved a game added by a register user. You could apply the appropriate filter and still order the results by the game name like this:

```
has_and_belongs_to_many :games, -> { where 'approved = true'.order('name ASC') }
```

Still other options are available; for instance you can constrain the columns selected in the associated objects using `select`:

```
has_and_belongs_to_many :games, -> { select('id, name') }
```

The ability to use a scope block in this manner is Rails 4-specific, however Rails 3 users aren't out of luck. Rails 3 users can modify the behavior of a `has_and_belongs_to_many` association by passing along an option rather than a scope block. For instance, if the user wanted a location's games to be sorted in ascending fashion by the game's name, the `order` option could be attached to the HABTM association like so:

```
has_and_belongs_to_many :games, :order => 'name asc'
```

Other options are also available to Rails 3 users for modifying the behavior. Logically you would only want to display those games having an `approved` attribute set to `true`. You can do that using the `:conditions` option:

```
has_and_belongs_to_many :games, :conditions => { :approved => true }
```

Useful HABTM Options

The `has_and_belongs_to_many` association supports a number of options that can change the association's behavior in very useful ways. I'll introduce the most commonly used options in this section, accompanied by several examples.

Changing the Child Class Name Rails eliminates much of the time and effort you'd have to spend making mundane decisions about application and database conventions by simply establishing a set of conventions for you. Consider for instance the following `has_and_belongs_to_many` association:

```
has_and_belongs_to_many :games
```

Rails will by default presume the associated model is `Game`. However, for a variety of reasons the class name might be different, `Juego` for instance (Spanish for “game”). You can override the default class naming convention using the `class_name` option:

```
has_and_belongs_to_many :games, class_name: 'Juego'
```

Changing the Join Table Name Active Record will presume the two models associated via `has_and_belongs_to_many` will be joined via a table named after the two model's underlying tables in alphabetical fashion with an underscore separating the two names. Therefore if you'd like to associate the `Location` and `Game` models, then the join table is expected to be named `games_locations`. You can change the default table name convention using the `join_table` option:

```
has_and_belongs_to_many :games, join_table: 'juegos_ubicaciones'
```

Changing the Foreign Key Just as Rails makes certain assumptions regarding the names of the `has_and_belongs_to_many` model tables, it also assumes the join table will include an attribute named using the parent model's name concatenated with `_id`. (for instance, `state_id`). If you'd like for some reason to override this convention, you can use the `foreign_key` option:

```
has_and_belongs_to_many :locations, foreign_key: 'ubicacion_id'
```

Introducing the `has_many :through` Association

The `has_and_belongs_to_many` association is useful when your *only* goal is to relate many records from one table with many records from another. While there's no question this feature has many purposes, you can quickly paint yourself into a corner. For instance what if you wanted to relate the `Game` and `Location` model using a `has_and_belongs_to_many` association, but additionally wanted to add a comment about the condition of the game at that particular location? The lack of a model dedicated to the joined data is sorely missed in this situation, given you could not for instance validate the comment attribute because it's not possible to define the validation! Not to mention it's not even possible to sanely update any attribute found in a `has_and_belongs_to_many` join table other than the foreign keys, because Active Record does not offer any native facility for doing so.

Fortunately, the Rails developers recognize that the capabilities of `has_and_belongs_to_many` are simply not suffice for many purposes. Should you require the join capabilities of `has_and_belongs_to_many` *alongside* all of the benefits that a model-driven association has to offer, you can take advantage of the `has_many :through` association. In this section I'll introduce you to this wonderful Active Record feature.

Creating a `has_many :through` Association

You'll interact with a `has_many :through` via an *independent model*, meaning you'll create this association much in the same way you'd create any other application model. Let's demonstrate this feature by implementing the aforementioned hypothetical involving including a comment attribute with each location/game relation. Begin by creating a new model. We'll call this model `Arcade` on the premise that each location houses games within a specially designated area. We'll refer to this special area as the *arcade*. This arcade could be a formally-defined area within a restaurant or a random corner of the laundromat. Remember, this `Arcade` model will tie the `Game` and `Location` models together, *and* manage any additional attributes and behaviors related to this association. Let's create the model now:

```
$ rails g model Arcade game:references location:references comment:string
  invoke  active_record
  create   db/migrate/20140702185142_create_arcades.rb
  create   app/models/arcade.rb
  invoke   test_unit
  create   test/models/arcade_test.rb
  create   test/fixtures/arcades.yml
```

The `game:references` and `location:references` statements are convenient shortcuts for `game_id:integer` and `location_id:integer`, respectively. Feel free to use whichever syntax suits your preference.

After generating the model, run the migration to create the corresponding `arcades` table. With the migration complete you'll find the following table in your application database:

```
mysql> describe arcades;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
game_id	int(11)	YES	MUL	NULL	
location_id	int(11)	YES	MUL	NULL	
comment	varchar(255)	YES		NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	

Next you'll need to define the association within the `Game` and `Location` models. Let's start with the `Game` model:

```
class Game < ActiveRecord::Base

  has_many :arcades
  has_many :locations, through: :arcades

end
```

This probably looks a bit odd, because you're defining not one but two `has_many` associations. First, you define the relation between the `Game` and `Arcade` models; the `Game` model `has_many` `Arcades` because a game can be found in more than one arcade. Next, we define the relation between the `Game` and `Location` model, telling Active Record we'd like to do this *through* the `Arcade` model. This is the magic that ensures the `Game/Location` association can be managed by way of the `Arcade` model.

Next let's define the `Location` model:

```
class Location < ActiveRecord::Base

  has_many :arcades
  has_many :games, through: :arcades

end
```

This definition looks almost identical to the association defined in the `Game` model, except that the `Location` model is defined as having many games through the `Arcade` model.

Finally, open up the `Arcade` model and just have a look at the association definitions (created automatically when the model was generated). There's nothing else we need to do with the model just yet, although we'll be returning to add validators in a bit. You'll see the `Arcade` model belongs to both the `Game` and `Location` models, because its companion table (`arcades`) will serve as the join table:

```
class Arcade < ActiveRecord::Base

  belongs_to :game
  belongs_to :location

end
```

We've completed the example `has_many :through` definition. Let's begin taking advantage of this new relationship!

Associating Objects Using the `has_many :through` Association

Presuming you've read the earlier introductions to the `has_many` and `has_and_belongs_to_many` associations, then understanding how to associate objects using `has_many :through` will be fairly straightforward. Like these other association types, you have a number of different options for creating new `has_many :through` associations; it's really up to you to choose the solution you feel most comfortable using. Because we're managing a location's games *through* the Arcade join model, you might presume you could create a new association using the `Location` model's `games` collection:

```
>> location = Location.find_by(name: 'Plain City Lanes & Pizza')
>> game = Game.find_by(name: 'Ms. Pac-Man')
>> location.games << game
>> location.games << Game.find(18)
Game Load (0.3ms)  SELECT `games`.* FROM `games` WHERE `games`.`id` = 18 LIMIT 1
T 1
(0.2ms)  BEGIN
SQL (0.4ms)  INSERT INTO `arcades` (`game_id`, `location_id`) VALUES (18, 16)
(0.3ms)  COMMIT
```

Sure enough it worked! Rails knows the relationship is being managed within the `arcades` table and so created a new record. However, this leaves the `comment` field blank and there's no straightforward way to set it via the `games` collection. Fortunately, there are alternative approaches involving the Arcade model:

```
>> location = Location.find_by(name: 'Plain City Lanes & Pizza')
>> game = Game.find_by(name: 'Ms. Pac-Man')
>> Arcade.create(game: game, location: location, comment: 'Average condition')
(0.2ms)  BEGIN
SQL (0.4ms)  INSERT INTO `arcades` (`comment`, `game_id`, `location_id`)
VALUES ('Average condition', 19, 1)
(0.5ms)  COMMIT
```

You could also reference the `Location` object's `arcades` collection's `create` method:

```
>> location = Location.find_by(name: 'Plain City Lanes & Pizza')
>> game = Game.find_by(name: 'Ms. Pac-Man')
>> location.arcades.create(game: game, comment: 'Average condition')
(0.1ms) BEGIN
SQL (0.5ms) INSERT INTO `arcades` (`comment`, `game_id`, `location_id`)
VALUES ('Average condition', 19, 1)
(1.1ms) COMMIT
```

Incidentally, because the association is bi-directional, you're not limited to creating the association via the `Location` model; you could also create it via the `Game` model:

```
>> location = Location.find_by(name: 'Plain City Lanes & Pizza')
>> game = Game.find_by(name: 'Ms. Pac-Man')
>> game.arcades.create(location: location, comment: 'Average condition')
```

Navigating a `has_many :through` Association

You navigate a `has_many :through` in exactly the same manner as was demonstrated in the earlier introductions to `has_many` and `has_and_belongs_to_many`. For instance, to iterate over all of the games associated with a location and which have an associated comment, you'll first retrieve the location and then iterate over its `arcades` array:

```
>> location = Location.find_by(name: '16-bit Bar')
>> location.arcades.where("comment <> ''").each do |arcade|
?> p "#{arcade.game.name} - #{arcade.comment}"
>> end
```

This will produce output that looks like this:

```
Zaxxon - Fantastic condition!
Tapper - Perfect game for a Barcade.
Joust - One of the classics!
```

Of course, you don't need to apply a filter when iterating over a `has_many :through` association; try removing `where("comment <> '')` to review the complete list of games available at this particular location.

There's a second way you could go about navigating an association without going through the `Arcade` model, done by defining what will be selected in the `has_many :through` association definition. For instance, if you wanted to access the comment directly through the `Location`'s games association, you could define the association like this:

```
class Location < ActiveRecord::Base
  has_many :arcades
  has_many :games, -> { select('games.*, arcades.comment as comment') }, :through => :arcades
end
```

By explicitly telling the model what you'd like to select, you can just access the `comment` attribute directly from the `Game` object:

```
>> location = Location.find_by(name: '16-bit Bar')
>> location.games.where("comment <> '').each do |game|
?> p game.comment
>> end
```

There are still other ways to simplify access to the join model data. For instance you could wrap the logic within an instance method, placing it within the `Location` or `Game` models. Spend some time experimenting with this and see what you come up with!

Deleting a has_many :through Association

You can remove a `has_many :through` association in a number of different ways. The most simplistic involves retrieving the desired `Arcade` object using the target `Game` and `Location` ids:

```
>> Arcade.find_by(game: Game.find(19), location: Location.find(8)).destroy
(0.1ms) BEGIN
SQL (0.9ms) DELETE FROM `arcades` WHERE `arcades`.`id` = 82
(0.1ms) COMMIT
```

However, this approach doesn't take advantage of the syntactical conveniences to be had by the association. It would be preferable to delete the relation while working with the `Location` object:

```
>> location.arcades.destroy(Arcade.find_by(location: location, game: game))
Arcade Load (0.6ms) SELECT `arcades`.* FROM `arcades`
WHERE `arcades`.`location_id` = 1 AND `arcades`.`game_id` = 22 LIMIT 1
(0.3ms) BEGIN
SQL (2.9ms) DELETE FROM `arcades` WHERE `arcades`.`id` = 147
(1.3ms) COMMIT
```

As you can see, `destroy` deleted the relationship outright. Let's run the same example again, but this time using `delete`:


```
>> location.arcades.delete(Arcade.find_by(location_id: location.id,
?>game_id: Game.find(16).id))
(0.1ms) BEGIN
SQL (0.3ms) UPDATE `arcades` SET `arcades`.`location_id` = NULL
WHERE `arcades`.`location_id` = 8 AND `arcades`.`id` IN (5)
(0.1ms) COMMIT
```

The `delete` method nullified the relationship, which would be useful in some situations but not for linking locations to games, particularly when comments are involved. So you'll want to be careful when choosing which method to use. Incidentally you can override the default behavior in a variety of ways; see the `dependent` option for more details.

Updating a Join Attribute

If the status of a game at some particular location changes, you'll logically want to change the associated comment. Per usual you can do so in a variety of ways, one of which involves finding the desired game within the location's arcade list, and using `update_attributes` to update the comment:

```
>> location = Location.find_by(name: '16-bit Bar')
>> location.arcades.find_by(game_id: 95).update_attributes(comment: 'New joystick\
k!')
(0.2ms) BEGIN
SQL (0.5ms) UPDATE `arcades` SET `comment` = 'New joystick!'
WHERE `arcades`.`id` = 69
(1.1ms) COMMIT
```

For most applications, understanding the syntax associated with updating an attribute found in a join table is somewhat moot, because of a great Rails feature that allows you to save associated attributes through the parent. I think this feature is best introduced in the context of forms, so I'll save this introduction for the next chapter.

Validating a `has_many :through` Association

When we generated the `Arcade` model you'll recall we adding a `comment` attribute in addition to the references to the `Game` and `Location` models. The `comment` attribute isn't required, but if it is added we want to keep the note brief. Let's add a validator for the `comment` attribute to limit the length to 30 characters:

```
class Arcade < ActiveRecord::Base

  belongs_to :game
  belongs_to :location

  validates :comment, length: { maximum: 30, message: 'The comment must be less \
than 30 characters.' }

end
```

Only a maximum length is imposed on the comment because we want the comment to be optional, meaning zero characters is fine. Let's work through an example that triggers the error:

```
>> location = Location.find_by(name: 'Plain City Lanes & Pizza')
>> game = Game.find_by(name: 'Ms. Pac-Man')
>> location.arcades.build(game: game, comment: 'Please read my super long commen\
t')
>> location.valid?
false
>> location.errors.full_messages
[
  [0] "Arcades comment A comment can't be longer than 30 characters"
]
```

Testing a has_many :through Association

To confirm a has_many :through association has been configured as desired, you can use the shoulda-matchers have_many and through matchers:

```
it 'should have many games through arcades' do
  expect(@location).to have_many(:games).through(:arcades)
end
```

Converting a has_and_belongs_to_many Association to has_many :through

Early on in ArcadeNomad's existence I used a has_and_belongs_to_many association to marry locations and their games, on the grounds I wasn't really interested in embellishing those relations with any additional information. However, while field testing the application it quickly became apparent that users might appreciate remarks about the condition of a location's games. For instance, I'd heard a local bowling alley had a Pac-Man game, and so swung by the location to confirm the matter, only to be disappointed that the upper left portion of the clear plastic monitor cover was so faded that it was effectively impossible to play the game! Of course, I still wanted to include the

bowling alley in the database, but attach a comment to the entry warning users of the flaw. To do so, I needed to convert the `has_and_belongs_to_many` association to `has_many :through` in order to be able to attach a comment attribute to each location/game relation.

Because it's entirely likely you'll encounter a similar situation with one of your future Rails projects, I thought it would be useful to include a section outlining the steps I took to complete this process. I began by creating the model used to manage the `has_many :through` relation. I'll call this model `Arcade` on the grounds that the locations managed in `ArcadeNomad` often have a designated location where the games are found, which would logically be referred to as the arcade:

```
$ rails g model Arcade game:references location:references comment:string
```

After running the migrations table the newly created `arcades` table will be populated and look something like this:

```
mysql> select * from arcades;
+-----+-----+-----+
| game_id | location_id | comment |
+-----+-----+-----+
|      5 |          9 | NULL    |
|      5 |         11 | NULL    |
|      7 |          8 | NULL    |
|      7 |         11 | NULL    |
|     16 |          8 | NULL    |
| ...    | ...        | ...     |
+-----+-----+-----+
```

After confirming that the data has been successfully migrated to the new table, I removed the `games_locations` table using a separate migration.

Next I added a length validator to the `Arcade` model's comment attribute, done by modifying the newly created `app/models/arcade.rb` class. Here's the revised model:

```
class Arcade < ActiveRecord::Base

  # A comment is optional but if provided shouldn't surpass 50 characters
  validates :comment,
    length: { maximum: 50,
              message: "A comment can't be longer than 50 characters" },
    allow_blank: true

  belongs_to :game
  belongs_to :location

end
```

Next, I modified the `Game` and `Location` models, converting the `has_and_belongs_to_many` associations to `has_many :through`. To do so, I replaced the `Game` model's `has_and_belongs_to_many :locations` declaration with this:

```
has_many :arcades
has_many :locations, through: :arcades
```

The `Location` model's `has_and_belongs_to_many :games` declaration was then replaced with this:

```
has_many :arcades
has_many :games, through: :arcades
```

With these changes saved, the migration from a `has_and_belongs_to_many` association to the more powerful `has_many :through` is complete!

Extending Active Record's Capabilities with Table Joins

Suppose you wanted to retrieve a set of states associated with locations categorized under Barcade. While you could use `find_by_sql` and craft a SQL query to get the job done, Active Record offers a more convenient syntax. Using the `joins` method, you can join the `locations` table to the `State` model and then apply a filter to the `locations` table:

```
>> states = State.select('distinct states.name').
?> joins(:locations).where('locations.category_id = ?', 4)
```

This retrieves a set of distinct `State` objects associated with locations assigned the category ID of 4, doing so by executing the following query:

```
SELECT distinct states.name FROM `states` INNER JOIN `locations`
  ON `locations`.`state_id` = `states`.`id` WHERE (locations.category_id = 4)
```

Be careful to note Active Record will by default use an `INNER JOIN`. While for this particular example an `INNER JOIN` is desired, there are many situations where you'll want to use a `LEFT JOIN`, because an `INNER JOIN` requires matches in both tables whereas a `LEFT JOIN` will retrieve *all* entries in the joined table regardless of whether there's a match. You can override this default behavior although doing so will require traipsing a bit further into the world of SQL:

```
>> states = State.select('distinct states.name').
?> joins('left join locations on states.id = locations.state_id')
?> .where('locations.category_id = ?', 4)
```

This produces the following query, retrieving all states assigned to *any* location, regardless of whether the location’s category_id is 4:

```
SELECT `states`.* FROM `states` left join locations
  on states.id = locations.state_id WHERE (locations.category_id = 4)
```

This returns an ActiveRecord::Relation which you can iterate over per usual:

```
states.each do |state|
  p state.name
end
```

You can perform joins using nested associations, allowing you to for instance filter on the categories table’s name attribute:

```
>> states = State.select('distinct states.name, states.id') \
?> .joins(locations: :category).where('categories.name = ?', 'Barcade')
```

This produces the following query:

```
SELECT distinct states.name, states.id FROM `states` INNER JOIN `locations`
  ON `locations`.`state_id` = `states`.`id` INNER JOIN `categories`
  ON `categories`.`id` = `locations`.`category_id`
WHERE ("categories.name = ?", 'Barcade')
```

Eager Loading of Associations

There’s a matter known as the “N + 1 Queries” problem that has long confused beginning Rails developers to the detriment of their application’s performance. To understand the nature of the issue, consider the following seemingly innocent query:

```
@locations = Location.limit(10)
```

In the corresponding application view you then iterate over the retrieved locations like so:

```
<ul>
  <% @locations.each do |location| %>
    <li><%= location.state.name %></li>
  <% end %>
</ul>
```

Pretty innocent bit of code, right? It certainly seems so until you realize these two snippets result in the execution of 11 separate queries! Thus the name “N + 1”, because we’re executing one query to retrieve the ten locations, and then 10 queries to retrieve the name of each location’s state name! To really drive this point home, here’s the eleven queries that executed when I ran the above two snippets:

```
SELECT `locations`.* FROM `locations` LIMIT 10
State Load (0.2ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.2ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.2ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.2ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.3ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.2ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.1ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.1ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.1ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 36 LIMIT 1
State Load (0.4ms) SELECT `states`.* FROM `states`
  WHERE `states`.`id` = 9 LIMIT 1
```

If you know you’re going to want to retrieve associated data, you can eager load the data using the `includes` method:

```
@locations = Location.includes(:state).limit 10
```

This produces just *two* queries, even after iterating over all ten locations just as we did before:

```
SELECT `locations`.* FROM `locations` LIMIT 10
State Load (4.1ms) SELECT `states`.* FROM `states` WHERE `states`.`id` IN (36,\
9)
```

Because the state name is already available to each record, there's no need to perform the additional queries!

Polymorphic Models

In addition to adding user registration and account management features to ArcadeNomad, I'd like to soon give users the ability to comment on games and locations. It would be fun to read nostalgic notes about their favorite 80's video game and reviews of a recent visit to one of the locations. To implement such a feature, one might presume we need to create separate comment models in order to both games and locations with their respective comments. This approach would however be repetitive because each model would presumably consist of the same data structure. You can eliminate this repetition using a *polymorphic association*.

Let's work through a simplified example that would use polymorphic associations to add anonymous commenting capabilities to the Game and Location models. Begin by creating a new model named Comment:

```
$ rails g model Comment body:text commentable_id:integer commentable_type:string
```

Next, open up the newly created Comment model (app/models/comment.rb) and edit it to look like this:

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

By declaring the model association as polymorphic, the model will look to the commentable_type and commentable_id attributes for the associated model type and the associated model record's primary key, respectively. Therefore if a comment were associated with a location having the record ID 112, then that record's commentable_type field would be set to Location and the commentable_id field would be set to 112. However, before this will work we need to complete configuration the association by modifying the Game and Location models. Open up these two models and add the following line alongside your other associations:

```
has_many :comments, :as => :commentable
```

Finally, run the migration and reload your console. With these changes in place, you can begin taking advantage of the polymorphic association. Let's try it out by attaching a comment to a game:

```
>> game = Game.find(1)
>> game.comments << Comment.create(body: "I love 1942!")
(0.2ms) BEGIN
SQL (0.5ms) INSERT INTO `comments` (`body`, `created_at`, `updated_at`)
VALUES ('I love 1942!', '2014-08-04 16:30:33', '2014-08-04 16:30:33')
(0.8ms) COMMIT
(0.3ms) BEGIN
SQL (0.7ms) UPDATE `comments` SET `commentable_id` = 1,
`commentable_type` = 'Game', `updated_at` = '2014-08-04 16:30:33'
WHERE `comments`.`id` = 1
(0.6ms) COMMIT
Comment Load (0.3ms) SELECT `comments`.* FROM `comments`
WHERE `comments`.`commentable_id` = 1 AND `comments`.`commentable_type` = 'Game'
```

Now let's attach a comment to a location:

```
>> location = Location.find(1)
>> location.comments << Comment.create(body: "My favorite bar!")
(0.2ms) BEGIN
SQL (1.7ms) INSERT INTO `comments` (`body`, `created_at`, `updated_at`)
VALUES ('My favorite bar!', '2014-08-04 16:28:25', '2014-08-04 16:28:25')
(0.2ms) COMMIT
(0.1ms) BEGIN
SQL (0.7ms) UPDATE `comments` SET `commentable_id` = 1,
`commentable_type` = 'Location', `updated_at` = '2014-08-04 16:28:25'
WHERE `comments`.`id` = 2
(0.2ms) COMMIT
```

With these two comments added, review your comments table and you'll see the following contents:

```
mysql> select * from comments;
+----+-----+-----+-----+-----+-----+
| id | body           | commentable_id | commentable_type | created_at | updated_at |
+----+-----+-----+-----+-----+-----+
| 1  | I love 1942!   | 1             | Game             | ...       | ...       |
| 2  | My favorite bar! | 1             | Location         | ...       | ...       |
```


Chapter 5. Mastering Web Forms

You're going to spend quite a bit of time building Rails applications that require various forms and models to work together in a seamless fashion. For instance a variety of forms are used by ArcadeNomad administrators to manage locations, categories and games, and by users to provide feedback and [get in touch with the ArcadeNomad staff](#)⁷⁷. While creating and integrating a simple form is easy enough, matters quickly become much more complicated for beginners when performing tasks such as populating select boxes, setting default values, or creating and processing forms that nest multiple models together within a single cohesive unit.

In this chapter I'll show you how to wield total control over your Rails-driven forms, covering a variety of form integration scenarios you're sure to encounter when the need arises to embed forms into your future applications.

Web Form Fundamentals

While all readers are quite familiar with web forms from the user's perspective, I'd imagine at least a few of you could benefit from a quick introduction to a few fundamental concepts developers need to keep in mind when integrating forms into a web application. If you're a knowledgeable web developer with plenty of experience working with web forms, then by all means skip ahead to the next section. I do however talk about a few Rails-specific form features in this section therefore if you don't have prior experience working with Rails-managed forms consider at least skimming the relevant parts of this section.

The following example is nearly identical to the form found on the [Contact ArcadeNomad](#)⁷⁸ page. It incorporates [HTML5-specific markup](#)⁷⁹, Rails-specific security features, and [Bootstrap-specific markup](#)⁸⁰ to produce a flexible, secure, and responsive form. Take a moment to examine the form markup before moving on to the dissection found below.

⁷⁷<http://arcadenomad.com/contact>

⁷⁸<http://arcadenomad.com/contact>

⁷⁹<http://diveintohtml5.info/forms.html>

⁸⁰<http://getbootstrap.com/>

```

<form accept-charset="UTF-8" action="/contact" class="new_contact_form"
  id="new_contact_form" method="post" role="form">
  <div style="display:none">
    <input name="utf8" type="hidden" value="â€œ">
    <input name="authenticity_token" type="hidden"
      value="V69+yh6Ndy2/NEby+LeTGiU">
  </div>

  <div class="form-group">
    <label for="contact_form_name">Name</label>
    <input class="form-control" id="contact_form_name"
      name="contact_form[name]"
      required type="text">
  </div>

  <div class="form-group">
    <label for="contact_form_name">E-mail Address</label>
    <input class="form-control" id="contact_form_email"
      name="contact_form[email]"
      required type="text">
  </div>

  <div class="form-group">
    <label for="contact_form_name">Message</label>
    <textarea class="form-control" id="contact_form_message"
      name="contact_form[message]"
      required></textarea>
  </div>

  <div class="form-group">
    <input class="btn btn-primary" name="commit" type="submit"
      value="Send message">
  </div>
</form>

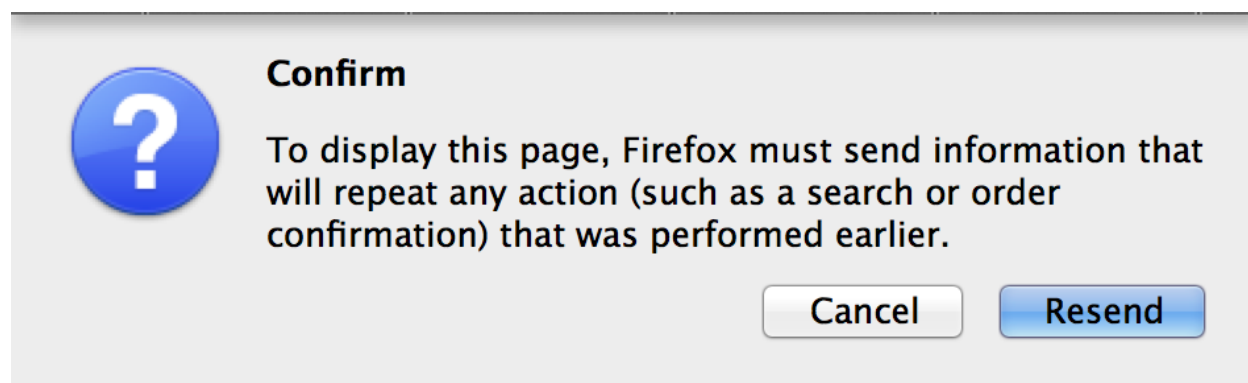
```

Let's review the pertinent characteristics of this form, beginning with the form enclosure. The form tag encloses the fields and other markup comprising the form:

```
<form accept-charset="UTF-8" action="/contact" class="new_contact_form"
      id="new_contact_form" method="post" role="form">
  ...
</form>
```

Let's review the relevant form attributes:

- The `accept-charset` attribute defines the character encoding intended to be used in conjunction with the form, should your intent be to override the default character set defined in the page header. The `action` attribute defines where the user will be taken when the form is submitted. Logically this should match a route definition found in your projects `routes.rb` file. Thanks to Rails' powerful route customization capabilities, there are some pretty nifty things you can do in regards to defining these endpoints, a topic I'll devote to a later section.
- The `method` attribute defines *how* the data will be sent to the destination. You might be familiar with the most common methods `GET` and `POST`, but not understand the important difference between the two. The short answer is that you should use `GET` for "safe" tasks, and `POST` for those deemed to be "unsafe". Safe tasks are those that can be repeatedly executed without negative consequences. For instance, you would typically use `GET` in conjunction with a search form, because search results could be cached, bookmarked and shared without negatively affecting the site nor the users. The `POST` method should be used for tasks that if executed more than once would pose a problem, such as sending a support request or charging a credit card. You've likely at one point or another mistakenly attempted to resend a `POST` form and been greeted with a message such the one presented by Firefox:



A Firefox POST warning

However, if you reload Google search results, you'll receive no such warning. This is because browser developers are aware of this important difference between `GET` and `POST`, and build in this warning as a cautionary step for users *should the website developer not have taken additional steps to prevent unwanted consequences* should a `POST` form be submitted more than once. Incidentally, even experienced developers might be surprised to know several other form methods exist! I'll talk all about these alternatives in the section, "Rails Routing and Forms".

- Finally, the `role` attribute is an [ARIA landmark](http://www.w3.org/WAI/GL/wiki/Using_ARIA_landmarks_to_identify_regions_of_a_page)⁸¹ used to define a particular region of the page, which is in this case a form.

Enclosed in a `div` styled as `display:none` you'll find the following two input tags:

```
<div style="display:none">
  <input name="utf8" type="hidden" value="â€œ">
  <input name="authenticity_token" type="hidden"
    value="V69+yh6Ndy2/NEby+LeTGiU">
</div>
```

Let's review the relevant aspects of this snippet:

- The tag named `utf8` assigned the name `utf8`⁸² and assigned the UTF-8-encoded check mark character ensures that the form values are submitted as UTF-8.
- The `authenticity_token` field is a Rails-specific security feature that prevents [cross-site request forgery \(CSRF\)](http://en.wikipedia.org/wiki/Cross-site_request_forgery)⁸³ by storing the randomly generated value in the field and also in a session. When the form is submitted, that field's value is compared with the session stored on the client to ensure they match; lacking such a feature it would be possible for third parties to attempt to forge form input values and send them directly to the server, potentially altering or destroying application data.

The three input tags assigned the `form-control` class are used to gather the user's name, e-mail address and message:

```
<input class="form-control" id="contact_form_name" name="contact_form[name]"
  required type="text">
...
<input class="form-control" id="contact_form_email" name="contact_form[email]"
  required type="text">
...
<textarea class="form-control" id="contact_form_message"
  name="contact_form[message]" required></textarea>
```

Let's review the pertinent aspects of these tags:

- All three fields are identified as `required`. This is an HTML5 feature that when implemented by browsers will automatically provide client-side validation without additional work on the part of the developer. I recommend you use this feature *in addition to* server-side validation.

⁸¹http://www.w3.org/WAI/GL/wiki/Using_ARIA_landmarks_to_identify_regions_of_a_page

⁸²<http://en.wikipedia.org/wiki/UTF-8>

⁸³http://en.wikipedia.org/wiki/Cross-site_request_forgery

- Each field uses a naming convention resembling a hash (`contact_form[name]`, `contact_form[email]`, and `contact_form[message]`). This is to facilitate easy retrieval of the form values subsequent to form submission. As you'll soon learn the creation of these field names can be automated by Rails.

With this general overview complete, I'll spend the remainder of this chapter discussing forms integration from the perspective of a Rails developer, drawing upon numerous live examples found in the *ArcadeNomad* code to illustrate various concepts.

Rails Routing and Forms

Before jumping into the any code used for generating and processing forms, let's take a moment to talk about some important plumbing-related matters pertinent to how Rails routes are typically configured in a manner that allows for sane management of the form request and response process. You might be aware that Rails espouses a concept known as *RESTful routing*. Rails' RESTful routing implementation is intended to facilitate the integration of the HTTP protocol's various *methods* within the framework of a Rails application controller. You've already been introduced to two of the HTTP methods, GET and POST, however several others exist, such as PUT, PATCH and DELETE. Rails embraces the power of HTTP methods by giving you the option of mapping these verbs to various actions defined within a controller. Simply defining a particular controller as an application *resource* will result in the presumption that seven different controller actions and corresponding URL endpoints exist. Consider for instance if we were to define the following resource within the Rails application's `routes.rb` file:

```
resources :games
```

Once defined, the Rails application will automatically begin routing specific requests to specific actions found in the `games` controller. I'll enumerate those requests and their corresponding actions in the following table.

HTTP Verb	Path	Controller Action	Description
GET	/games	games#index	Present a list of games
GET	/games/new	games#new	Present an HTML form for creating a game
POST	/games	games#create	Create a new game
GET	/games/:id	games#show	Present a specific game
GET	/games/:id/edit	games#edit	Present an HTML form for editing a game
PATCH/PUT	/games/:id	games#update	Update a specific game
DELETE	/games/:id	games#destroy	Delete a specific game

Resource routes are a prime example of Rails' pledge of supporting the notion of [convention over configuration](http://en.wikipedia.org/wiki/Convention_over_configuration)⁸⁴, because they can ensure uniformity within controllers responsible for defining an application's CRUD (Create, Retrieve, Update, Delete) operations. For instance once you've determined the need to create a resourceful route, you can take two easy steps towards implementation of the route by first creating a controller containing seven actions:

```
$ rails g controller games index show new create edit update destroy
```

This will generate a controller named `games_controller.rb` that looks like this:

```
class GamesController < ApplicationController

  def index
  end

  def show
  end

  def new
  end

  def create
  end

  def edit
  end

  def update
  end

  def destroy
  end

end
```

After creating the controller, open the `config/routes.rb` file because we'll need to replace the route definitions created by Rails when the RESTful controller was generated. Given all of the attention the Rails project showers on the importance of RESTful routing it is a mystery to me why they don't define the resourceful route by default. However it's easy to remedy this shortcoming by replacing the newly added routes with the resource definition. In this case find the following lines:

⁸⁴http://en.wikipedia.org/wiki/Convention_over_configuration

```
get 'games/index'  
  
get 'games/show'  
  
get 'games/new'  
  
get 'games/create'  
  
get 'games/edit'  
  
get 'games/update'  
  
get 'games/destroy'
```

Once found, *delete* these lines, replacing them with:

```
resources :games
```

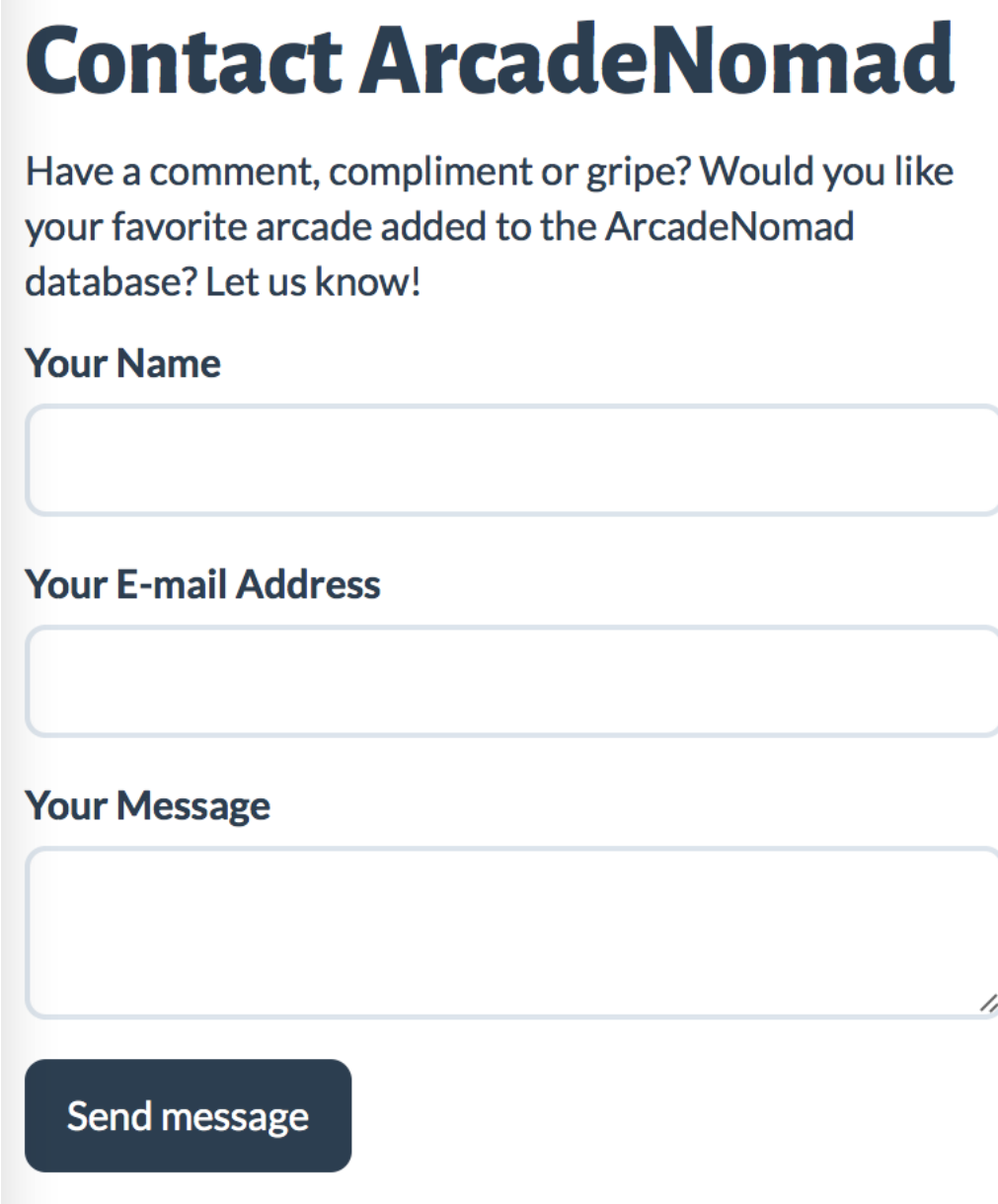
After saving the changes, try navigating to <http://0.0.0.0:3000/games> and <http://0.0.0.0:3000/games/new> and you'll be greeted with the `games#index` and `games#new` action view placeholders, respectively. You were routed to these actions because your requests both employed the GET method and requested access to the `/games` and `/games/new` paths, respectively.

Although later in this chapter I'll work through a detailed example involving how to create the forms used to manage ArcadeNomad's locations and games, let's take this opportunity to talk about the typical patterns you'll want to embrace when creating forms for adding and updating model records. Returning to the table outlining the resourceful route definitions, you'll see that GET `/games/new` and POST `/games` are used to access the games controller's `new` and `create` actions, respectively. The `new` action is primarily responsible for housing the form used to input the new game record's attributes. When submitted, the form data will be sent to `/games` using the POST method, meaning the data will be processed by `games#create`. If any validation errors are uncovered, the user will be *returned* to `games#new` and prompted to resolve the issues, otherwise the new record will be added to the database. Once the record has been successfully saved, it is up to the developer whether to return the user to `games#new` to add another game (and include a notification informing the user of the record being successfully added), redirect the user to the newly added game, or perform some other task altogether.

Editing existing records follows a similar pattern, with the `games#edit` and `games#create` actions used in conjunction with one another to load the desired record into a form for editing, and then update the modified record attributes within the database.

Creating a Simple Contact Form

Because I'd imagine a fair number of readers have no prior experience creating a proper Rails-powered form, let's kick things off by creating a simple contact form that ArcadeNomad users can use to contact yours truly. This support form is located at <http://arcadenomad.com/contact>⁸⁵ and consists of three fields, including the user's name, email address, and message (see below figure).

The image shows a web form titled "Contact ArcadeNomad" in a large, bold, dark blue font. Below the title is a paragraph of text: "Have a comment, compliment or gripe? Would you like your favorite arcade added to the ArcadeNomad database? Let us know!". The form consists of three input fields: "Your Name" (a single-line text box), "Your E-mail Address" (a single-line text box), and "Your Message" (a multi-line text area). At the bottom of the form is a dark blue button with the text "Send message" in white. The entire form is set against a light gray background.

Contact ArcadeNomad

Have a comment, compliment or gripe? Would you like your favorite arcade added to the ArcadeNomad database? Let us know!

Your Name

Your E-mail Address

Your Message

Send message

ArcadeNomad's contact form

⁸⁵<http://arcadenomad.com/contact>

The URL endpoint `http://arcadenomad.com/contact` is actually handled by the About controller's `new` and `create` actions (`new` presents the form via the GET method and `create` processes it via POST). You're of course free to manage the contact feature within any controller you please, however because the About controller (which also houses the "About ArcadeNomad" page at <http://arcadenomad.com/about>⁸⁶) would otherwise not use the `new` and `create` actions I decided to consolidate the contact feature there. To manage this feature in the About controller while using the convenient `/contact` path you'll need to define two aliases in the `config/routes.rb` file:

```
match '/contact', to: 'about#new', via: 'get', as: 'contact_new'
match '/contact', to: 'about#create', via: 'post', as: 'contact'
```

Next you'll need to generate the About controller if you haven't already done so. The following example creates the controller, and adds the `index`, `new`, and `create` actions:

```
$ rails g controller about index new create
```

With the controller, actions, and views generated you can proceed with writing the code used to implement the feature.

Installing and Configuring the MailForm Gem

There are plenty of reasons why you would want to send an e-mail through a Rails application, and in fact the task is so central to web development that Rails natively supports the capability via [Action Mailer](#)⁸⁷. While you could absolutely use Action Mailer when the need arises to send form data via e-mail, I prefer to use the [MailForm gem](#)⁸⁸, created by [Jose Valim](#)⁸⁹ and [Carlos Antonio](#)⁹⁰. MailForm was created expressly for this purpose and therefore makes it easy to add the feature while not sacrificing on reliability and testing. To install MailForm, add the following line to your project Gemfile:

```
gem 'mail_form'
```

Save the changes and run `bundle install` to install the gem.

⁸⁶<http://arcadenomad.com/about>

⁸⁷http://guides.rubyonrails.org/action_mailer_basics.html

⁸⁸https://github.com/plataformatec/mail_form

⁸⁹<http://github.com/josevalim>

⁹⁰<http://github.com/carlosantoniodasilva>

Creating the ContactForm model

MailForm is so convenient because it is built atop both the [ActiveModel](#)⁹¹ and [ActionMailer](#)⁹² modules, allowing you to take advantage of great features such as validations and mail delivery. In fact, we're going to create a model that represents the data submitted through the form. But rather than inherit from ActiveRecord::Base, this model will instead inherit from MailForm::Base. Here's the model used in ArcadeNomad:

```
class ContactForm < MailForm::Base

  attribute :name, :validate => true

  attribute :email, :validate => /\A([\w\.\%\+\-]+)@([\w\-\]+\.\.)([\w]{2,})\z/i

  attribute :message, :validate => true

  # Declare the e-mail headers
  def headers
    {
      :subject => 'ArcadeNomad - Contact Request',
      :to       => 'arcadenomad@example.com',
      :from     => 'jason@example.com'
    }
  end
end
```

This class defines three attributes, name, email and message, validating name and message for presence and using a regular expression to validate the email address. The headers method exists simply to return a hash containing information that will be passed into the e-mail header, such as the e-mail subject and sender. The hash containing this information supports any of the headers supported by the [ActionMailer mail method](#)⁹³; I'm just keeping it simple for the sake of example.

Speaking of ActionMailer, you'll also need to configure Action Mailer so it knows how to deliver the mail. You can include this configuration in the appropriate environment file (development.rb, production.rb, etc). The following example shows you how to configure Action Mailer to use your Gmail address, keeping in mind you would need to replace `replace_this_username` with your Gmail username and additionally update the password:

⁹¹<http://api.rubyonrails.org/classes/ActiveModel/Model.html>

⁹²http://guides.rubyonrails.org/action_mailer_basics.html

⁹³<http://api.rubyonrails.org/classes/ActionMailer/Base.html#method-i-mail>

```
config.action_mailer.delivery_method = :smtp

config.action_mailer.smtp_settings = {
  address:           'smtp.gmail.com',
  port:              587,
  domain:            'gmail.com',
  user_name:         'replace_this_username@gmail.com',
  password:          'secret',
  authentication:    'plain',
  enable_starttls_auto: true
}
```

Save these changes and then returning to the `ContactForm` model, save it as `contact_form.rb` within the `app/models` directory. Of course, you'll need to change the subject and to values to suit the purpose of your own application. Save the changes and test out the model by firing up the Rails console:

```
>> contact = ContactForm.new(:name => 'Nolan', :email => 'nolan@example.com',
  ?> :message => 'I love ArcadeNomad!')
>> contact.valid?
=> true
>> contact.deliver
```

Check the inbox associated with the e-mail address you assigned to the `to` option and you should see an e-mail message containing the provided name, e-mail address and message! Believe it or not, it's that easy to send an e-mail through `MailForm`. Of course, we have some more work to do before this feature can be considered complete, so let's get to it!

Updating the new Action

We'll next need to integrate the `ContactForm` model into the new action. Believe it or not we only need to add a single line to the method:

```
def new

  @contact = ContactForm.new

end
```

Simple enough; we instantiate the `ContactForm` model, creating a new instance variable named `@contact`. But why do we need an instance variable of type `ContactForm`? I'll solve the mystery next.

Creating the Contact Form

Earlier in this chapter I showed you the *rendered* form HTML, introducing several key Rails- and HTML5- related form features in the process. Note my emphasis on *rendered* because you won't actually hand-code the form! Instead, you'll use Rails' fantastic form generation capabilities to manage this tedious task for you. Below I've pasted in the section of code found in ArcadeNomad's app/views/about/new.html.erb view that's responsible for generating the rendered contact form HTML:

```
<%= form_for @contact, url: contact_path, html: { role: 'form' } do |f| %>

  <div class="form-group">
    <%= f.label :name, 'Your Name' %>
    <%= f.text_field :name, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.label :email, 'Your E-mail Address' %>
    <%= f.text_field :email, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.label :message, 'Your Message' %>
    <%= f.text_area :message, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.submit 'Send message', :class => 'btn btn-primary' %>
  </div>

<% end %>
```

If this is your first encounter with the `form_for` helper then I'd imagine this example looks a tad scary. However once you build a few forms in this fashion I promise you'll wonder how you ever got along without it. Let's break down the key syntax used in this example:

- The `form_for` method is a *form helper* that binds a form to a model object. This is convenient because we want to bind the `ContactForm` model to the contact form fields in order to properly validate the input. It accepts as its first argument an instance variable, in this case `@contact`. The options hash that follows defines other form characteristics, such as the action and attributes such as `class` and `role`. The `form_for` helper yields an object used to subsequently create the form controls, which in this example is `f`. You probably noticed there is no reference

to the form method in this example; this is because the default is POST and therefore I've not included the option. If you'd like to override the default you can pass along `method: :get`.

- Inside the `form_for` block you'll see a series of methods are executed to generate the various fields comprising the form. This is a relatively simplistic form therefore only a few of the available field generation methods are used, including `label` (for creating form field labels), `text_field` (for creating form text fields), `text_area` (for creating a form text area), and `submit` (for creating a submit button). Note how the `text_field` and `text_area` methods all accept as their first argument a model attribute name (`name`, `email`, and `message`, respectively). All of the methods also accept an assortment of other options, such as class names and HTML5 form attributes.

Add this code to your project's `app/about/new.html.erb` file and navigate to the view via `http://0.0.0.0:3000/about/new.html` (or `http://0.0.0.0:3000/contact` if you added the route alias) and you should see the same form as that found at `http://arcadenomad.com/contact!`



Rails also offers another form helper called `form_tag`⁹⁴. However in almost every case I prefer to use the model-backed `form_for` helper and therefore I've opted to not discuss it in this book.

Updating the Create Action

Finally, we need to update the About controller's `create` action so the submitted form contents can be processed:

```
def create

  @contact = ContactForm.new(params[:contact_form])

  if @contact.deliver

    flash.now[:success] = 'Thank you for your message!'

  else

    flash.now[:error] = 'Could not send message.'
    render :new

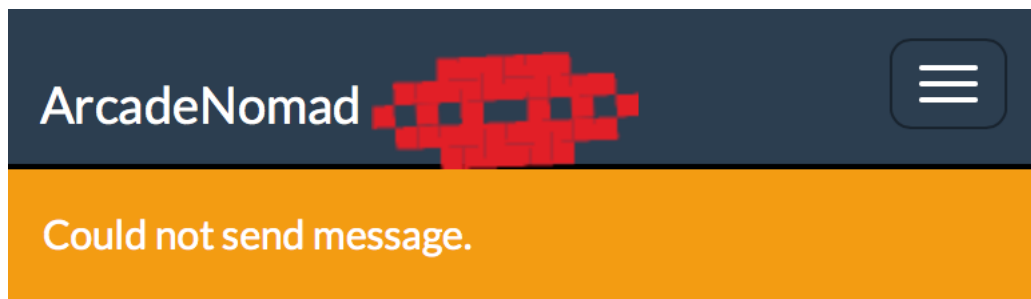
  end

end
```

end

⁹⁴http://guides.rubyonrails.org/form_helpers.html

The create action begins by populating the instance variable `@contact` with an initialized `ContactForm` object. This object contains the values of the name, email, and message fields found in the contact form. Next, we attempt to execute the `deliver` method (inherited by `ContactForm` model via `MailForm::Base`). If successful, a flash message of type `success` is generated prior to rendering the create action's view (`app/views/about/create.html.erb`). If the `deliver` method fails, a flash message of type `error` is generated and the new action's view is instead rendered. For instance when there is an issue sending the e-mail through `ArcadeNomad`'s contact form, the flash message is displayed directly below the header as depicted in the following screenshot.



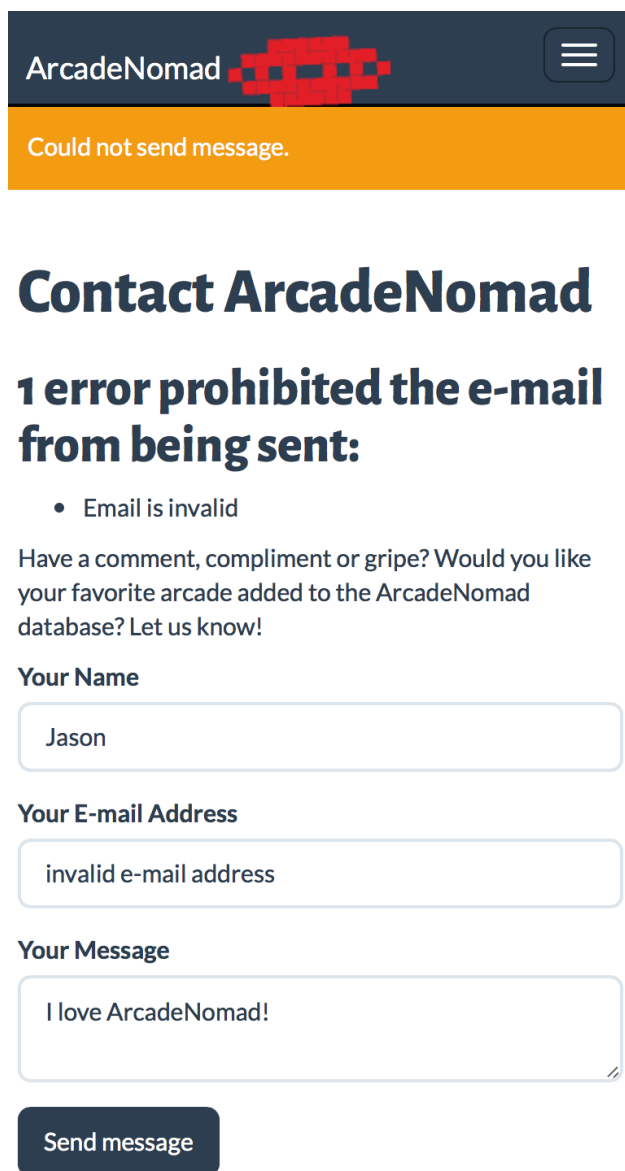
Contact ArcadeNomad

Displaying a Flash Message

I'll talk more about the convenience of using flash messages in the next section.

Should `deliver` fail, it's presumably because the provided data was not valid per the validation definitions found in the `ContactForm` model and therefore you want to give the user an opportunity to correct the error by returning to the form. By assigning the submitted form data to the `@contact` instance variable, the `form_for` helper found in `app/views/about/new.html.erb` will *populate* the form fields, saving the user the hassle of entering the data anew!

In addition to populating the form with the supplied data, we'll also want to provide the user with specific information about the errors. You can do this by displaying the validation errors within the view, as depicted in the below screenshot.



The screenshot shows the top of a web application. A dark blue header contains the text "ArcadeNomad" and a red pixelated logo. Below the header is an orange banner with the text "Could not send message." in white. The main content area has a large heading "Contact ArcadeNomad" and a subheading "1 error prohibited the e-mail from being sent:". Below this is a bulleted list with one item: "Email is invalid". A paragraph of text follows: "Have a comment, compliment or gripe? Would you like your favorite arcade added to the ArcadeNomad database? Let us know!". There are three input fields: "Your Name" with the value "Jason", "Your E-mail Address" with the value "invalid e-mail address", and "Your Message" with the value "I love ArcadeNomad!". At the bottom is a dark blue button labeled "Send message".

Displaying Contact Form Validation Errors

Next I'll show you how to both generate flash messages and display validation errors.

Displaying Flash Messages and Errors

[Flash messages](#)⁹⁵ are a great Rails feature that you can use to notify users of various task outcomes such as the contact form submission status. Flash messages are particularly convenient because they are intended to live for at most just one additional request. Flash messages are native to Rails, and setting them within an action is easy. For instance the following example sets a flash notifying the user of successful logout:

⁹⁵http://guides.rubyonrails.org/action_controller_overview.html#the-flash


```
flash[:notice] = 'You have successfully logged out'  
redirect_to root_url
```

Within the corresponding view you can iterate over any existing flash messages:

```
<% flash.each do |name, msg| %>  
  <%= content_tag :div, msg, class: name %>  
<% end %>
```

This snippet will iterate over the flash messages, creating output that looks like this:

```
<div class="notice">  
  You have successfully logged out  
</div>
```

Thus the flash type (in this case, `notice`) serves a useful purpose in that it can be used in conjunction with CSS stylization to present the flash message in an appealing and attention-grabbing fashion. Also, the Rails documentation identifies `alert` and `notice` as two supported flash types but in actuality you can identify any key you please when creating a new flash message. For instance the following is perfectly valid:

```
flash[:successful_logout] = 'You have successfully logged out'  
redirect_to root_url
```

You might have noted I've taken special care to include a `redirect_to` statement following the flash creation statements. This is because flash messages are by default available on the *next* request, and therefore by redirecting to the desired destination that newly created message will be made available. However, as you have a particularly keen eye you probably also noticed I used `flash.now` when generating the flash messages in the earlier `create` action. This is because in both branches of the conditional statement I am *rendering* a view rather than redirecting to one. By rendering rather than redirecting to a view there is no additional request and therefore the flash message would not yet be available to the rendered view. Rails offers the `flash.now` alternative for situations where you would like to make the flash message available to the *current* request and thus the reason why I used it in the earlier example.

Displaying Validation Errors

When notifying users of problems related to forms you'll likely want to supplement the flash messages with some additional detailed information about precisely what caused the error to occur. Fortunately you can easily pass the validation error messages into the view because as you might recall from Chapter 2 they are accessible via the model object. The following snippet is used within the ArcadeNomad About controller's new view to detect and output validation messages:

```
<% if @contact.errors.any? %>
  <div>
    <h2><%= pluralize(@contact.errors.count, 'error') %>
      prohibited the e-mail from being sent:</h2>

    <ul>
      <% @contact.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

This snippet is pretty straightforward, first determining if any errors exist using Ruby’s `any?` method. If yes, the number of errors are determined using the `count` method, and that number is passed into the `pluralize` view helper in order to determine whether the singular “error” or plural “errors” should be used when constructing the `h2` text. Finally, the `@contact` object’s `errors` array is iterated over, with each error message inserted into a bulleted list.

Testing the Contact Form

Testing the contact form is somewhat more involved than the tests you’ve encountered throughout the book, because we’re interested in testing far more than just the `ContactForm` model. We’d also like to test the web form, and additionally ensure the e-mail is sent as expected. To do so, we’ll need to perform some additional test-related configuration, beginning with the installation of [Capybara](http://jnicklas.github.io/capybara/)⁹⁶.

Capybara is a Ruby gem that simulates user interaction with a web application. You can incorporate Capybara commands into your tests, using these commands to access views, click links, confirm the presence of text, and even complete forms. With some additional configuration out of the scope of this book, you can even use Capybara to interact with remote APIs and test Javascript-driven features (see the [Capybara documentation](http://rubydoc.info/github/jnicklas/capybara)⁹⁷) for more information about these additional features). For the purposes of the examples found in this chapter we’ll stick to using Capybara’s fundamental but still incredibly useful features. With that said, let’s install Capybara by adding the following line to your `Gemfile`:

```
gem 'capybara'
```

After saving the changes, run `bundle install` to install the gem. Next, open up `spec/rails_helper.rb` and add the following line:

⁹⁶<http://jnicklas.github.io/capybara/>

⁹⁷<http://rubydoc.info/github/jnicklas/capybara>

```
require 'capybara/rails'
```

Save the changes and you're ready to begin writing tests using Capybara!

Writing the Integration Specs

To test the contact form and `ContactForm` model we're going to write a few *integration specs*. Integration specs are a tad more involved than the model specs we've been writing so far, because the intent is to ensure that multiple application components are working together as expected. Capybara's ability to interact with your web application much in the same manner as a user greatly reduces the amount of work you'd otherwise have to do when testing component integration manually. Let's write a few tests to ensure the form can be successfully submitted, and that the validation errors are presented to the user should the supplied name, email or message not meet the specifications.

By default, Capybara expects the specs to reside in a directory named `features` found in the project's spec directory, so create that directory now. Next, let's create a spec that confirms when the user submits valid form data, the user will see the desired flash message `Thank you for your message!`. Create a file named `about_spec.rb` and add the following contents to it, saving the file to the newly created `features` directory:

```
require 'rails_helper'

feature 'Visitor uses contact form' do

  scenario 'to submit a valid contact form' do
    visit contact_new_path
    fill_in 'contact_form_name', :with => 'Nolan Bushnell'
    fill_in 'contact_form_email', :with => 'nolan@example.com'
    fill_in 'contact_form_message', :with => 'I love ArcadeNomad!'
    click_button 'Send message'
    expect(page.body).to have_content('Thank you for your message!')
  end

end
```

With the changes saved, run the spec in the usual manner:

```
$ rspec -fd spec/features/about_spec.rb
```

You should see the following output:

```
Visitor uses contact form
  to deliver a valid contact form
```

```
Finished in 0.23422 seconds (files took 1.05 seconds to load)
1 example, 0 failures
```

This spec looks quite different from those we’ve created so far. We’re taking advantage of both Capybara and RSpec to interact with the `/about/contact` view, actually filling out the form in the same way a typical user would, clicking the submit button, and then confirming the destination view contains the text, `Thank you for your message!`. Because this flash message only appears when MailForm’s `deliver` method is successful, one could presume the test can reasonably confirm the contact action, corresponding view, `ContactForm` model, and MailForm gem are working together as desired. There is however another step we can take to be even more certain this feature is working as expected; I’ll return to this matter in a moment.

The previous spec reasonably confirms our expectation when the user *successfully* completes the form. But what about those instances where he forgets to input his name or provides an invalid e-mail address? We can right specs to confirm the validators are implemented as desired. Let’s create a spec to confirm the appropriate error message is displayed when the user neglects to provide an e-mail address. Add the following test directly following the previous, making sure it is placed inside the feature block:

```
scenario 'but provides a blank e-mail address' do
  visit contact_new_path
  fill_in 'contact_form_name', :with => 'Nolan Bushnell'
  fill_in 'contact_form_message', :with => 'I love ArcadeNomad!'
  click_button 'Send message'
  expect(page.body).to have_content("Email can't be blank")
end
```

This scenario is identical to the previous except no e-mail address is provided, presumably causing the e-mail to be invalidated and error message, “Email can’t be blank” to be presented to the user. Run the spec in the usual manner:

```
$ rspec -fd spec/features/about_spec.rb
```

Presuming the test passes, this time you’ll see You should see the following output:

```
Visitor uses contact form
  to deliver a valid contact form
  but provides a blank e-mail address
```

```
Finished in 0.58545 seconds (files took 3.05 seconds to load)
2 examples, 0 failures
```

Take some time to become more familiar with Capybara's basic syntax by writing specs to confirm users must provide a valid name and message.

Testing Mail Delivery

When the user successfully completes and submits the contact form, it's presumed the e-mail is successfully generated and sent to the configured recipient address. However in the previous successfully validated spec we only confirmed the user is presented with a message indicating a successful outcome, without actually confirming the e-mail was actually sent. [RailsCasts' Episode #275 \(How I Test\)](http://railscasts.com/episodes/275-how-i-test?view=asciicast)⁹⁸ offers a great way to confirm that ActionMailer is actually delivering the e-mail by having a look at `ActionMailer::Base.deliveries`. In addition to confirming ActionMailer's delivery mechanism was successful, we can confirm the e-mail's headers such as the sender and recipient. To do so, create a file named `mailer_macros.rb` within the `spec/support` directory and add the following code to it:

```
module MailerMacros
  def last_email
    ActionMailer::Base.deliveries.last
  end

  def reset_email
    ActionMailer::Base.deliveries = []
  end
end
```

The `last_email` method returns the most recently delivered e-mail, which presumably would be the e-mail most recently sent as a result of running a spec. The `reset_email` will clear out the delivered e-mails so we can be sure of a known state when running specs involving the `ActionMailer::Base.deliveries` array.

Next, open up `spec/rails_helper.rb` and inside the `RSpec.configure` block add the following lines:

⁹⁸<http://railscasts.com/episodes/275-how-i-test?view=asciicast>

```
config.include(MailerMacros)
config.before(:each) { reset_email }
```

These two lines include the newly created `MailerMacros` module, and then runs the `reset_email` method prior to each spec in order to ensure the `ActionMailer::Base.deliveries` array is cleared out. Next, we'll modify the earlier spec used to confirm successful submission of a valid contact form to additionally examine the e-mail sent via the `MailForm` gem:

```
scenario 'to submit a valid contact form' do
  visit contact_new_path
  fill_in 'contact_form_name', :with => 'Nolan Bushnell'
  fill_in 'contact_form_email', :with => 'nolan@example.com'
  fill_in 'contact_form_message', :with => 'I love ArcadeNomad!'
  click_button 'Send message'
  expect(page.body).to have_content('Thank you for your message!')
  expect(last_email.to).to have_content('arcadenomad@wjgilmore.com')
  expect(last_email.from).to have_content('nolan@example.com')
end
```

This spec is identical to the original version, except for the newly added last two lines. These lines confirm the most recently delivered e-mail (checked by executing the `last_email` method found in the `MailerMacros` module) has its to and from headers set to `arcadenomad@wjgilmore.com` and `nolan@example.com`, respectively. Of course you're free to confirm other characteristics of the delivered e-mail, such as the subject and body content.



You'll logically want to visually confirm the contents of an automatically generated e-mail by having it delivered to your inbox the first few times you test the feature. However presuming your test suite will likely run hundreds and even thousands of times over the course of a project, it makes little sense to have to constantly delete these e-mails from your inbox. You can instead configure the [MailCatcher](https://github.com/sj26/mailcatcher)⁹⁹ gem to “capture” these e-mails and view their contents via a web interface.

Managing Your Application Data

Although the current game plan is for `ArcadeNomad` to eventually offer features for allowing community-driven addition and editing of arcade locations and games, for reasons of expediency the current release's data is curated by a team of internationally renowned retro arcade experts (okay it's just me). Interested contributors can of course add candidate locations using the public form located

⁹⁹<https://github.com/sj26/mailcatcher>


at <http://arcadenomad.com/locations/new>¹⁰⁰, however those locations do not appear on the site until they're approved via a password-protected administration console. In this section I'll talk about the various interfaces used to add and manage ArcadeNomad's data, along the way introducing you to numerous intermediate and advanced aspects of Rails application form integration.


The Location Submission Form

ArcadeNomad users can submit new locations for consideration using the form located at <http://arcadenomad.com/> (see below figure). The form includes fields for the location's name, address, a brief description, and of course for identifying the games found at the location.

¹⁰⁰<http://arcadenomad.com/locations/new>

¹⁰¹<http://arcadenomad.com/locations/new>

ArcadeNomad



Suggest a Location

We've approximated your current position using geolocation. If you would like to clear the form, press the Clear button at the bottom of page.

Name

Must be unique. e.g. BW3's Hilliard

Description

A sentence or two describing this arcade

Street

7140 Wellington Court

City

Dublin

State

Ohio

Zip

43016

Category

Airport

Games at this Location

Select games

1942
After Burner II
After Burner

Despite consisting of several additional fields, this form isn't really any more complicated than the contact form, although it does introduce two new concepts. You'll notice in the screenshot that the form incorporates two select boxes, one for assigning a location category (Barcade, Airport, Skating Rink, etc.), and another for identifying the games found at that location. Let's focus on how these two fields are implemented and saved along with the other form fields.



Address Geocoding

Because most users will presumably interact with this form using a mobile device, the [Google Maps Geocoding API](https://developers.google.com/maps/documentation/geocoding/)¹⁰² is used to prepopulate the form with the address associated with the user's location in order to reduce the amount of typing required to complete the form (and therefore hopefully reduce the typing errors commonly associated with using the cramped mobile phone keyboard). I won't go into any details regarding how geocoding is integrated into ArcadeNomad, however if you purchased the ArcadeNomad code and want to learn more, review the `app/assets/javascripts/locations.js.coffee` file for the code used to geolocate the user.

The form is created using the `form_for` helper just as the earlier contact form was. The object yielded by `form_for` is `f`, and in the below example this object is used to create the category field's label and drop-down list:

```
<div class="form-group">
  <%= f.label 'Category' %>
  <%= f.collection_select :category_id, Category.order('name asc'), :id, :name, \
  {},
  { :class => 'form-control', :id => 'location-category-id' } %>
</div>
```

The `collection_select` method is notoriously confusing to beginners, and admittedly it tripped me up on numerous occasions early on in my Rails career. However, its syntax will make sense if we just take the time to introduce each input parameter:

- The first parameter (`:category_id`) is used as the “key” portion of the select field's name attribute. In this case, once the field is rendered the field name will be `location[category_id]`.
- The second parameter (`Category.order('name asc')`) retrieves the values used for the select field's option values and text. Although you could pass in a query's results from the controller, it's also possible to execute the query directly within a view. I almost never do this, with the exception being when using `collection_select`. In this case we're retrieving all of the categories found in the `categories` table, ordering the results by the category name in ascending fashion.

¹⁰²<https://developers.google.com/maps/documentation/geocoding/>

- The third parameter (`:id`) identifies the model attribute used to populate the select field's option value attributes.
- The fourth parameter (`:name`) identifies the model attribute used to populate the select field's option text.
- The fifth parameter, which in this case is blank, is used to define any field options such as whether a prompt should be used. If you did want to specify a prompt, you would define this parameter as `{prompt: 'Select Category'}`.
- The sixth and final parameter defines any HTML options. In this case we're defining two; the field's CSS class (`form-control`) and the ID (`location-category-id`).

Once rendered to the browser, the drop-down HTML will look something like this:

```
<select id="location-category" class="form-control" name="location[category_id]">
<option value="12">Airport</option>
<option value="1">Amusement</option>
<option value="11">Amusement Park</option>
...
<option value="9">Pool Hall</option>
<option value="2">Restaurant</option>
<option value="6">Skating Rink</option>
</select>
```

Provided the field name corresponds to the attribute used to cement the association (in the case of the `Location` object you might recall from Chapter 4 that it is `category_id`), and provided you updated the appropriate white list to include the `category_id` field (see the Chapter 2 section, “Introducing Strong Parameters”, if this matter of whitelisting doesn’t make any sense), then the `belongs_to` association will be created right alongside the other location record attributes!

Updating the `has_many :through` Association

The other association we need to handle in the location submission form is matching the selected games to the location. Despite the need to pass an array of values (game IDs) rather than just a single value such as a category ID, this is really not any more difficult to implement because Rails practically all of the work for you. You will however need to make a simple modification to the `Game` and `Location` model association declarations defined in the last chapter, adding the `inverse_of` option to each. Here’s the modified `Game` model association:

```
class Game < ActiveRecord::Base

  has_many :arcades, inverse_of: :game
  has_many :locations, through: :arcades
  accepts_nested_attributes_for :arcades

end
```

And here's the modified Location model association:

```
has_many :arcades, inverse_of: :location
has_many :games, through: :arcades
accepts_nested_attributes_for :arcades
```

You won't typically see the `inverse_of` option in tutorials, and strictly speaking it isn't always required, however adding it better informs Rails as to the nature of the association when two models define the same relationship from opposite directions, and is useful when nesting multiple models within the same form. Thoughtbot's Pat Brisbin published a blog post on this topic last year titled [accepts_nested_attributes_for with Has-Many-Through Relations](http://robots.thoughtbot.com/accepts-nested-attributes-for-with-has-many-through-relations)¹⁰³ that is well worth reading as it explains in detail the reasoning behind including this option.

Here's the HTML and Rails helpers used to create the multiple select field:

```
<div class="form-group">
  <%= f.label 'Select Games' %>
  <%= f.collection_select :game_ids, Game.all, :id, :name, {},
    { :multiple => true, :class => 'form-control', :id => 'location-game-ids' } %>
</div>
```

This HTML is practically identical to that used for the category, except in the HTML options parameter (the sixth parameter) we're declaring the field to allow multiple selections, because logically a location may host more than one retro arcade game. When rendered to the browser the multi-select field HTML will look like this:

¹⁰³<http://robots.thoughtbot.com/accepts-nested-attributes-for-with-has-many-through>

```
<select id="location-game-ids" class="form-control"
  name="location[game_ids][]" multiple="multiple">
  <option value="1">1942</option>
  <option value="2">After Burner II</option>
  <option value="3">After Burner</option>
  <option value="4">Airwolf</option>
  ...
  <option value="123">Xybots</option>
  <option value="124">Zaxxon</option>
  <option value="125">Zero Wing</option>
</select>
```

Like the category field, provided you've properly defined the field name (in this case by identifying `game_ids` as the first parameter) and properly updated your strong parameter declaration, Active Record will happily do the rest of the work for you. Be sure to refer back to Chapter 2's "Introducing Strong Parameters" section (or have a look at the `ArcadeNomad` code), because you need to declare array-based parameters a tad differently within the strong parameter declaration than you would others for this to work properly.

After the user has successfully submitted the suggested arcade location, it's up to the administrator to review and approve the contribution. This is accomplished through the `ArcadeNomad` administration console. I'll show you how this is created next.

The ArcadeNomad Administration Console

Logically a Rails application's layout and functionality are going to play a major role in its success, but above all it's the quality of the data that is going to truly endear users to your project. Therefore it's crucial to ensure your application data is always in tip-top shape. One convenient way to do this is through a web-based administration console. For instance via `ArcadeNomad`'s administration console I can manage categories, games and locations. Of course, this is a pretty simplistic console, serving a suitable role for demonstrating the concepts found in this section. These consoles can become much more involved depending upon the project's size and scope. For instance, I was recently involved in a project that hosted an administration console used to manage almost 50 different data sets, including user accounts and various application settings!

Creating an Administration Namespace

An administration console will serve as a central location for managing multiple aspects of the application, including data and perhaps general behavior via a set of administrative preferences. Given such complexity it's practically an application unto itself, consisting of multiple controllers and views. There are a few different ways you could go about organizing the code used within the administration area however in my opinion the best approach involves creating an administration namespace. You can do so by prefixing the name of a controller with `admin/`, like this:

```
$ rails g controller admin/index index
create  app/controllers/admin/index_controller.rb
route   namespace :admin do
  get   'index/index'
end
invoke  erb
create  app/views/admin/index
create  app/views/admin/index/index.html.erb
invoke  test_unit
create  test/controllers/admin/index_controller_test.rb
invoke  helper
create  app/helpers/admin/index_helper.rb
invoke  test_unit
create  test/helpers/admin/index_helper_test.rb
invoke  assets
invoke  coffee
create  app/assets/javascripts/admin/index.js.coffee
invoke  scss
create  app/assets/stylesheets/admin/index.css.scss
```

Carefully examine the output from the above command, and you'll see all of the usual files are nested inside a directory named `admin`! You can visit the newly created index controller's index view by navigating to `http://0.0.0.0/admin/index/index`, where you'll see the standard placeholder text found in the generated view. Of course, this route is really ugly so let's fix it by making a change to `routes.rb`. Open `routes.rb` and find the following lines:

```
route namespace :admin do
  get 'index/index'

end
```

Replace this with:

```
namespace :admin do
  root 'index#index'
  resources :index
end
```

The root `'index#index'` statement tells Rails to consider the Index controller's index action found within the `admin` namespace to be the default destination when navigating to the `admin` namespace's root directory, meaning with the changes saved you can reach the `admin/index` controller's index action by navigating to `http://0.0.0.0:3000/admin/`.

I use this `index` action as a basic menu for navigating to different administrative interfaces. It's just a collection of links, and looks like the screenshot presented below.



Administration Console

Manage Categories

- Add a Category
- Edit Categories

Manage Games

- Add a Game
- Edit a Game

Manage Locations

- Edit Locations
- Add a Location

ArcadeNomad's administrative home page

Mind the Namespaces

Open up the newly created Index controller and you'll see the following code:

```
class Admin::IndexController < ApplicationController
  def index
  end
end
```

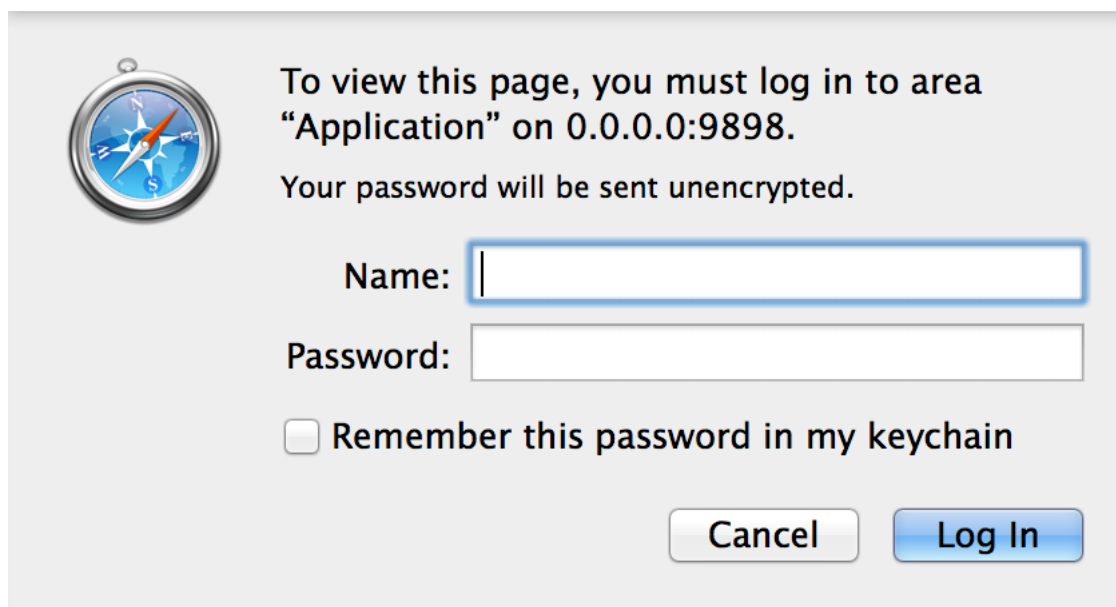
Carefully note the controller namespacing (Admin)! If you were to remove or edit this namespace, the controller will no longer work as intended. Other than this difference, your controllers, views and helpers behave as you'd expect for any Rails application, and require no further configuration beyond making sure the relevant routes are properly defined in `routes.rb`.

Restricting Access

Logically you'll want to restrict access to the administration console. Rails offers plenty of different ways to authenticate users, however ArcadeNomad's needs aren't particularly exotic and so I've opted for what is possibly the simplest approach that nonetheless is flexible enough to protect specific controllers and even specific actions. Rails has long offered a little-known method called `authenticate_or_request_with_http_basic`¹⁰⁴ that wraps HTTP's basic access authentication¹⁰⁵ scheme. When properly integrated into your application, users attempting to access the protected endpoints will be prompted to provide a username and password via a popup login window such as the one presented in the below screenshot.

¹⁰⁴<http://api.rubyonrails.org/classes/ActionController/HttpAuthentication/Basic/ControllerMethods.html>

¹⁰⁵http://en.wikipedia.org/wiki/Basic_access_authentication



Requiring authentication

Believe it or not, you can integrate this feature into your application in three simple steps. First, open up your project's `app/controllers/application_controller.rb` file and add the following method:

```
protected
def authenticate
  authenticate_or_request_with_http_basic do |username, password|
    username == ADMIN_USERNAME && password == ADMIN_PASSWORD
  end
end
```

This method calls the `authenticate_or_request_with_http_basic` method, passing in the username and password provided via the login window. If the username matches the constant `ADMIN_USERNAME` and the password matches the constant `ADMIN_PASSWORD` (more on these constants in a moment), `true` is returned, otherwise `false` is returned. By adding the `authenticate` method to the Application controller it will be made accessible to all of the other application controllers. We'll return to exactly how this method is used in other controllers in a moment.

The constants used for comparing the supplied username and password are made available via an initialization file named `admin_credentials.rb`, placed in your project's `config/initializers` directory. The file's contents look like this:

```
ADMIN_USERNAME = 'admin'
ADMIN_PASSWORD = 'ohsosecret'
```


Any file placed in the `config/initializers` directory is considered an *initializer*, meaning anything contained in the file will execute following initialization of the Rails framework and loading of any gems, thereby making configuration settings, constants, etc. available, including the two constants we defined in `admin_credentials.rb`. While you certainly could embed the credentials as strings within the `authenticate` method, doing so would result in the credentials being tracked in your code repository, something which is generally not an ideal practice. Therefore presuming you use an initialization file to manage these credentials, you'll want to make sure the file is ignored by your repository. For instance if you're using Git you can update the `.gitignore` file by adding the following line to it:

```
config/initializers/admin_credentials.rb
```

Finally, you'll add the `authenticate` method to any controller you'd like to protect by forcing it to execute using `before_filter`:

```
class Admin::IndexController < ApplicationController

  before_filter :authenticate

  def index
  end

end
```

Once the filter has been added, restart your server and try to visit the protected controller. If everything is properly configured you'll be presented with the login window displayed in the previous screenshot.

Bear in mind HTTP basic access authentication does *not* guarantee security in a production environment, because the credentials are not encrypted when passed from the browser to server! To ensure the credentials are passed without the possibility of interception by a nosy third party you'll want to use an SSL certificate in conjunction with the administration console. This topic is out of the scope of the book however there are plenty of online resources that can help you add an SSL certificate to your server.

Managing Categories

Of the three types of data managed within the administration console (categories, games and locations), categories are the easiest to manage because the only editable attribute is the category name meaning we only really need to keep our eye on misspelled categories. Of course we might occasionally wish to add a category as well, but it's highly unlikely we'd want to delete a category (and therefore either disable or reassign any locations associated with the deleted category), meaning the administration console's category-related interfaces are limited to adding new categories and editing category names. Because the immediate need only requires the editing and creation of categories, we can create a limited set of RESTful actions:

```
$ rails g controller admin/categories index show new create edit update
```

Don't forget to update the `config/routes.rb` file to take advantage of RESTful routing. Below I've added the line `resources :categories` to the `administration` namespace declaration created earlier in this section:

```
namespace :admin do
  root 'index#index'
  resources :categories
  resources :index
end
```

Rather than rehash the boilerplate process of creating a new record, a topic already covered in earlier sections, let's focus on the matter of editing a category. This involves the newly created `Categories` controller's `edit` and `update` methods, however you'll typically arrive at the `edit` method by clicking on a link or some other visual element found in a category listing, such as the one presented in the below screenshot:

ArcadeNomad



Categories

Tracking 13 categories.

Airport

Amusement

Amusement Park

Arcade

Bar

Barcade

Bowling Alley

Laundromat

Movie Theater

Other

Pool Hall

Restaurant

Skating Rink

In this screenshot, each category is linked to the edit action, with the category's ID passed along to the edit action can subsequently retrieve the appropriate category for populating the edit form. For instance the link used to edit one of the categories found in the ArcadeNomad database looks like `http://arcadenomad.com/admin/categories/10/edit`. This category list is generated in the Category controller's index action. Because there's a limited number of categories I didn't even bother with pagination, instead simply retrieving all of the categories and ordering by the name attribute in ascending fashion:

```
class Admin::CategoriesController < ApplicationController

  def index
    @categories = Category.order('name asc')
  end

end
```

The line in the associated index view used the appropriate [RESTful link helper](#)¹⁰⁶ to create the edit links such as the one found in the previous paragraph:

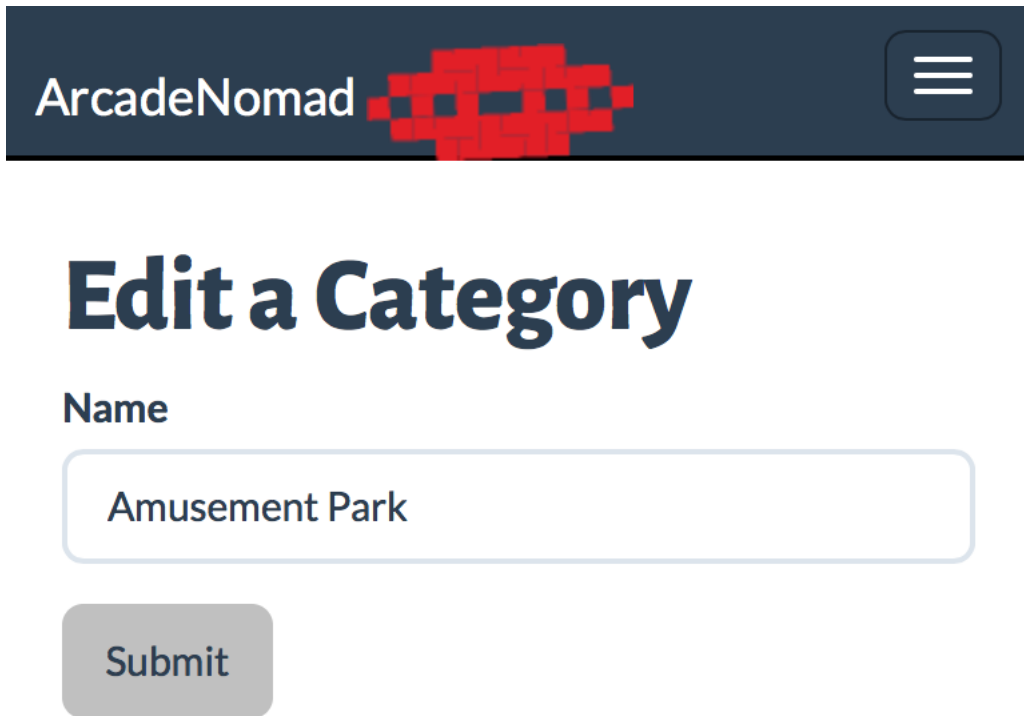
```
<%= link_to category.name, edit_admin_category_path(category) %>
```

Next, we need to create the edit action code. This too is pretty simple, as we're just passing the provided ID into the `find` method:

```
def edit
  @category = Category.find(params[:id])
end
```

With the edit action in place, we can create the corresponding view, used to generate the form presented in the following screenshot:

¹⁰⁶<http://guides.rubyonrails.org/routing.html#path-and-url-helpers>



ArcadeNomad

Edit a Category

Name

Submit

Editing an ArcadeNomad category

The relevant part of the edit view is presented here:

```
<h1>Edit a Category</h1>

<% if @category.errors.any? %>
  <div>
    <h2><%= pluralize(@category.errors.count, 'error') %> prohibited this
      category from being updated:</h2>

    <ul>
      <% @category.errors.messages.values.each do |message| %>
        <li><%= message[0] %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<%= form_for [:admin, @category], :html => {'role' => 'form'} do |f| %>

  <div class="form-group">
    <%= f.label :name %>
```

```

    <%= f.text_field :name, :placeholder => 'Must be unique. e.g. Bowling Alley',
      :class => 'form-control' %>
  </div>

  <div class="form-group">
    <button type="submit" class="btn">Submit</button>
  </div>

<% end %>

```

The syntax used in the above snippet is by now familiar to you, although pay close attention to the `form_for` parameters because a tiny adjustment had to be made in order to account for the admin namespacing. Rather than just specifying the model object (`@category`), you'll need to additionally pass along the desired namespace, which in this case is `:admin`, meaning the first parameter looks like `[:admin, @category]`.

OK, just one more step. We next need to create the update action:

```

def update

  @category = Category.find(params[:id])

  if @category.update(category_params)
    flash[:notice] = 'Category successfully updated.'
    redirect_to edit_admin_category_path(@category.id)
  else
    render 'edit'
  end

end

...

private
def category_params
  params.require(:category).permit(:name)
end

```

In the update method we kick things off by again retrieving the target category, and then calling the update method, passing along the whitelisted parameters as defined by the `category_params` method. If you don't understand why this approach is being taken be sure to read the section "Introducing Strong Parameters", located in Chapter 2.

Managing Locations

By now I'd imagine you're really starting to get the hang of what's required to carry out CRUD tasks (Create, Retrieve, Update and Delete) within your Rails applications, having learned how to create new locations and subsequently how to edit categories. Therefore much of the ArcadeNomad administration console's location management facilities isn't going to strike you as particularly interesting. So what I instead would like to focus on in this section is a powerful Active Record feature that causes a great deal of confusion among even experienced users.

All of the forms we've created so far are fairly straightforward, involving either one model or marrying two models together via a predefined association such as `belongs_to` or `has_many :through`. But consider ArcadeNomad's `has_many :through` relationship shared by the `Game` and `Location` model. You might recall from Chapter 2 the section "Converting a `has_and_belongs_to_many` Relation to `has_many :through`" in which I talked about the reasons and process behind converting this association from a `has_and_belongs_to_many` to a `has_many :through`, the primary reason being that I wanted the option of adding a comment to each specific location/game association, for instance to denote damage to the game. Consider for a moment how an administration form might be structured so as to allow these comments to be updated, particularly in light of the fact the comments are *part of* the association. Therefore it only makes sense to edit these comments *in the context of* the location/game association. I do this using an interface that looks like that found in the below screenshot.

Telephone

614-921-9999

Url

http://www.buffalowildwings.com/en/locations

Category

Restaurant

Games at this Location

Galaga

Ms. Pac-Man / Galaga multicade.

Ms. Pac-Man

Ms. Pac-Man / Galaga multicade.

Submit

Clear

Editing a location and its associated games

This screenshot depicts the bottom half of the administration console's location edit form, found in `app/views/admin/locations/edit.html.erb`. As you can see, this particular location hosts two games, including *Ms. Pac-Man* and *Galaga*. However I thought it important to point out these two games are actually part of the popular *Ms. Pac Man/Galaga* multicade, meaning there's actually only

one arcade cabinet on the premises. Because the comments are specific to this particular relationship, it makes a lot of sense to manage them within the editorial view of this location. So how is this accomplished? It's easier than you think.

Let's begin with a review of the required changes to the model definitions. As mentioned, this is the administration console's `Location` administration form, meaning in addition to location-specific attributes we're nesting attributes found in another model, namely `Arcade`. In order for this to be possible we need to update the `Location` model's `has_many :through` association definition like so:

```
has_many :arcades
has_many :games, through: :arcades
accepts_nested_attributes_for :arcades
```

The standard `has_many :through` definition (comprised of the first two lines) remains in place, but now there's a third line stating that the `Location` model can accept nested attributes for (`accepts_nested_attributes_for`) the `Arcade` model. This statement is what allows for `Arcade` model attributes to be modified via a `Location` object.



If you wanted to modify `Arcade` model attributes via a `Game` object, you'd of course have to add the `accepts_nested_attributes_for` statement within the `Game` model definition. Further, you're not limited to using `accepts_nested_attributes_for` with solely `has_many :through`.

Next we'll focus on the form. The form found at `app/views/admin/locations/edit.html.erb` looks like the other forms introduced so far, except for the part below the header `Games at this Location`. That section looks like this:

```
<%= form_for [:admin, @location], :url => admin_location_path(@location.id),
  :html => {'role' => 'form'} do |f| %>

  ...

  <%= f.fields_for :arcades do |arcade| %>

    <div class="form-group">
      <%= arcade.label arcade.object.game.name, arcade.object.game.name %><br />
      <%= arcade.text_field :comment, :placeholder => 'Add comment here',
        :class => 'form-control input' %>
    </div>

  <% end %>
```

```
...  
  
<% end %>
```

When rendered to the browser, the HTML for each rendered arcade object looks like this:

```
<div class="form-group">  
  <label for="location_arcades_attributes_0_1942">1942</label>  
  <br>  
  <input id="location_arcades_attributes_0_comment" class="form-control input"  
    type="text" value="asdfblah" placeholder="Add comment here"  
    name="location[arcades_attributes][0][comment]">  
</div>  
<input id="location_arcades_attributes_0_id" type="hidden" value="83"  
  name="location[arcades_attributes][0][id]">
```

Note in particular how the primary key for each arcades record is automatically embedded into the form using a hidden field. This is important, because Rails will use that ID in order to perform the update. This means the linking table (arcades in this case) must use the default id column as its primary key rather than a composite key.

Returning to the `fields_for` example, the `fields_for` method creates a scope on the `f` object, giving you the opportunity to refer to the model identified as being accessible via the `accepts_nested_attributes_for` declaration. Within the block you can refer to the Arcade model's attributes in the same fashion as the object passed into the `form_for` block. You'll see for instance the comment text field is declared by referring to the arcade object's `text_field` method, and then passing along as the first parameter `:comment` because we'd like to edit the arcades table's comment field. Additionally, you'll see in this example an interesting tactic for retrieving the name of the arcade game in order to use the name as the field label. Note this isn't a feature limited to the `fields_for` block; you can also use the same tactic within a `form_for` block.

Finally, let's review the administration console's Locations controller's update method. Pay attention because this is the part that seems to cause the most confusion among developers. The update method looks identical to any you've seen so far:

```
def update
  @location = Location.find(params[:id])

  if @location.update(location_params)
    flash[:notice] = 'Location successfully updated.'
    redirect_to edit_admin_location_path(@location.id)
  else
    render 'edit'
  end
end
```

But here's the important part: the `update` method accepts the parameter whitelist, and this whitelist *must* include the attributes you'd like to update within the `fields_for` block, in addition to the hidden `id` attribute:

```
def location_params
  params.require(:location).permit(:name, ...,
    :arcades_attributes => [:id, :comment])
end
```

With these pieces in place, you can begin taking advantage of this fantastic feature!

Keep in mind the `accepts_nested_attributes_for` class method isn't limited to `has_many :through` associations, and frankly the example provided above is fairly simplistic in light of all the other cool things you can do with it. In a forthcoming update to this book I'll be sure to expand coverage of this powerful feature to other association types. However, this example should be enough to help you sort out those aspects of nested forms that tend to be the source of most confusion.

Useful Links Be sure to peruse the following link for a complete summary of what's available to the `fields_for` method:

- [The `fields_for` API documentation](http://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html#method-i-fields_for)¹⁰⁷

Useful Forms-Related Gems

I've no doubt more experienced Rails developers are apoplectic over my decision to base the examples found in this chapter on Rails' native form builder features, because there are a number of third-party form builders that many consider to be far superior alternatives. Frankly I agree, and will introduce two particularly popular alternatives in this section, however I think it would be rather

¹⁰⁷http://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html#method-i-fields_for

presumptive of me to focus on one specific alternative without even bothering to introduce what's available natively to Rails. At the same time, it would be careless to not at least offer a cursory introduction to a few other options (including one that is so convenient you don't even have to build any administrative forms!). In this concluding section I'll try to have my cake and eat it too by introducing a few of these third-party solutions.

Better Forms with Formtastic

Created by [Justin French](https://github.com/justinfrench)¹⁰⁸, [Formtastic](https://github.com/justinfrench/formtastic)¹⁰⁹ is a very popular alternative to Rails' native form building capabilities, largely because it does a lot of the heavy lifting for you in terms of authoring and styling forms. Exactly how much work does it do for you? Recall the contact form presented in the earlier section, "Creating the Contact Form":

```
<%= form_for @contact, url: contact_path, html: { role: 'form' } do |f| %>

  <div class="form-group">
    <%= f.label :name, 'Your Name' %>
    <%= f.text_field :name, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.label :email, 'Your E-mail Address' %>
    <%= f.text_field :email, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.label :message, 'Your Message' %>
    <%= f.text_area :message, :class => 'form-control', :required => true %>
  </div>

  <div class="form-group">
    <%= f.submit 'Send message', :class => 'btn btn-primary' %>
  </div>

<% end %>
```

This form can be rendered practically identically using the following Formtastic syntax:

¹⁰⁸<https://github.com/justinfrench>

¹⁰⁹<https://github.com/justinfrench/formtastic>

```
<%= semantic_form_for @contact, {url: contact_path, html: { role: 'form' }} do |\
f| %>
  <%= f.input :name, :label => 'Your Name' %>
  <%= f.input :email, :label => 'Your E-mail Address' %>
  <%= f.input :message, :as => :text, :label => 'Your Message' %>
  <%= f.action :submit, :label => 'Send message', :button_html => {class: 'btn\
  btn-primary'} %>
<% end %>
```

As you can see from this simple example, Formtastic will attempt to determine the desired field type (text, check box, radio, textarea, etc.) based on the underlying attribute type. Therefore name and email are automatically rendered as text fields, but I overrode the message attribute (by passing along `:as => :text` because despite being a varchar column I want it to render as a text area.

Also, I cheated a bit with this example in order to ensure it and the original contact form rendered identically; Because ArcadeNomad relies heavily on Bootstrap due to my horrible design skills, this example actually uses [Matthew Bellantoni¹¹⁰](https://github.com/mjbellantoni/formtastic-bootstrap)'s [Formastic Bootstrap¹¹¹](https://github.com/mjbellantoni/formtastic-bootstrap) gem, which behaves practically identically to Formtastic except that Formtastic Bootstrap adheres to Bootstrap conventions for form layout.

Because I'm only offering a cursory overview of Formastic, it doesn't make sense to provide detailed installation instructions so instead be sure to have a look at the project [README¹¹²](#) for all of the details. If you want to take advantage of Bootstrap styling, have a look at the [Formtastic Bootstrap¹¹³](#) README after installing Formtastic.

Simple Form

[Simple Form¹¹⁴](#) is another very popular form builder replacement. Created and maintained by [Plataformatec¹¹⁵](#), the Simple Form developers aim to stay true to the project's name by offering an easy yet capable form generator that aids in the generation of form components while leaving layout-related matters largely up to you. As an example let's reconstruct the contact form using Simple Form:

¹¹⁰<https://github.com/mjbellantoni/>

¹¹¹<https://github.com/mjbellantoni/formtastic-bootstrap>

¹¹²<https://github.com/justinfrench/formtastic>

¹¹³<https://github.com/mjbellantoni/formtastic-bootstrap>

¹¹⁴https://github.com/plataformatec/simple_form

¹¹⁵<http://plataformatec.com.br/>

```

<%= simple_form_for(@contact, {url: contact_path,
                               html: {class: 'form', role: 'form' },
                               defaults: {
                                 input_html: {class: 'form-control' },
                                 wrapper_html: {class: 'form-group' } }
                               }) do |f| %>

  <%= f.input :name, label: 'Your Name' %>
  <%= f.input :email, label: 'Your E-mail Address' %>
  <%= f.input :message, as: :text, label: 'Your Message' %>
  <%= f.button :submit, class: 'btn btn-primary' %>

<% end %>

```

Like Formastic, Simple Form will attempt to select the appropriate form control based on the attribute type. This works fine for the name and email controls, however in the example I've overridden the message control using the as attribute. I also saved a few keystrokes by defining several control defaults in the simple_form_for constructor.

You might have noticed this example looks pretty similar to that used to introduce Formastic, and for good reason: Simple Form's DSL was based on Formtastic's! Therefore transitioning from one to the other is pretty easy.

Again, lacking a significant section on this topic I'll instead just point you to the project [README](#)¹¹⁶ for the latest instructions. The installation process is practically identical to Formtastic, and because Simple Form offers Bootstrap integration you won't have to worry about dealing with an additional third-party gem.

Easy Administration with Active Admin

Like ArcadeNomad, most web applications consist of both a public and private interface, with the latter used to manage various aspects of the application such as comment approval, blog entry creation and editing, and model content management (such as arcade game descriptions and location addresses). If you lack the time and resources required to build a proper administrative console then check out the fantastic [ActiveAdmin](#)¹¹⁷ gem. Not only can ActiveAdmin save you and your clients an extraordinary amount of time building these administrative interfaces, but thanks to its fantastic design, the aesthetics won't suffer one iota. For instance, check out the slick default interface Active Admin offers for editing an ArcadeNomad game:

¹¹⁶https://github.com/plataformatec/simple_form

¹¹⁷<http://www.activeadmin.info/>

Dev Arcadenomad Com Dashboard Admin Users Comments **Games** admin@example.com Logout

ADMIN / GAMES / ZAXXON /

Edit Game

Manufacturer	<input type="text" value="Sega"/>
Name*	<input type="text" value="Zaxxon"/>
Description*	<input type="text" value="A vintage arcade game."/>
Release date	<input type="text" value="1982"/>
Slug	<input type="text" value="zaxxon"/>

Editing a game with Active Admin

One aspect of Active Admin that I find really attractive is its flexible configuration. Unlike other similar solutions, Active Admin isn't an "all or nothing" solution in the sense you can select specifically which models you'd like to manage in the Active Admin interface, done using a simple Rake task. It also offers a powerful search interface which proves handy when the number of records grows into the hundreds and even thousands (see below screenshot).

Filters

ARCADES

Any

LOCATIONS

Any

MANUFACTURER

Any

NAME

Contains

DESCRIPTION

Contains

CREATED AT
 -

UPDATED AT
 -

RELEASE DATE

Equals

SLUG

Contains

Filter

Clear Filters

Active Admin's powerful search interface



If you're in need of a turnkey administration console such as Active Admin also be sure to have a look at [RailsAdmin](#)¹¹⁸.

The [ActiveAdmin website](#)¹¹⁹ offers installation instructions, although be forewarned, at the time of this writing Rails 4 support was still in development and I found the documentation to be a tad outdated which caused a few initial hiccups. Check out the [Easy Active Record blog](#)¹²⁰ as I plan on writing a detailed post outlining installation instructions.

Conclusion

Wow this was a pretty long chapter, but I hope the work put into it helps you to grasp what is an often confusing and even infuriating aspect of Rails development. Even despite the length, there's plenty left to cover regarding Active Record and forms, and I'll be sure to regularly update both the book and the blog in order to illustrate other concepts. In the meantime, let's move on to the last chapter, in which we'll talk about debugging, optimizing, and securing your Rails application.

¹¹⁸https://github.com/sferik/rails_admin

¹¹⁹<http://activeadmin.info/documentation.html>

¹²⁰<http://easyactiverecord.com/blog>

Chapter 6. Debugging and Optimizing Your Application

While most developers don't find debugging and optimization to be particularly enthralling tasks, the long-term success of your project is going to depend in no small part upon how effectively you intertwine these activities alongside the far more exciting matter of feature development. Yet far too many developers tend to employ unproductive debugging and optimization techniques, often waiting until they've been burned a time or two before getting serious about following best practices. I think this largely has to do with the errant belief the associated tools and processes are too difficult to successfully grasp in such a way that they can be painlessly integrated into the workflow. Not so! Historically this has certainly been the case, but in recent years a great deal of effort has been put into building user-friendly tools and services that dramatically improve your ability to debug and optimize web applications. In this chapter I'll introduce you to a few of my favorite solutions, focusing of course on how they relate to Rails applications, and specifically to Active Record.



I've no doubt I'll receive a fair bit of grief over not having covered this particular gem or that particular technique. In this first release I'm just trying to cover some of what I believe to be the most fundamental debugging- and optimization-related utilities outside of those introduced in the Rails documentation and in basic tutorials. If you have a suggestion then by all means e-mail me and I'll certainly try to cover the tool in a future release.

Debugging Your Application

No matter whether your application is in the early prototyping stage or has long since shifted into maintenance mode, you're going to be devoting time and effort to debugging. Because of this inescapable reality, honing efficient debugging skills can save you literally hundreds of hours over the course of a project, not to mention eliminate a lot of unnecessary grief. So for those of you whose debugging skills leave something to be desired, this section may well wind up being the most important in the entire book! Mind you this isn't an introduction to debugging Rails applications; if you're completely new to the topic then I suggest starting by reading the section of the Rails manual titled “[Debugging Rails Applications](http://guides.rubyonrails.org/debugging_rails_applications.html)”¹²¹. Instead I'll focus on a few concepts and tools that I regularly put into practice to boost my own productivity.

¹²¹http://guides.rubyonrails.org/debugging_rails_applications.html

Introducing Awesomeprint

Throughout this book we spent quite a bit of time inside the Rails console; the no-frills interface is unparalleled for experimenting with and peering into all aspects of your application, Active Record objects included. The lack of frills can admittedly become a tad tedious though when peering into larger objects, as demonstrated by this example:

```
>> g = Game.find(124)
Game Load (0.1ms) SELECT `games`.* FROM `games`
  WHERE `games`.`id` = 124 LIMIT 1
=> #<Game id: 124, name: "Zaxxon", description: "A vintage arcade game.",
    created_at: "2014-07-21 15:42:21", updated_at: "2014-07-21 15:42:21",
    release_date: 1982, manufacturer_id: 2, slug: "zaxxon">
```

Fortunately, there's a fantastic gem called [Awesome Print](https://github.com/michaeldv/awesome_print)¹²² that allows you to have your console cake and eat it too, automatically applying formatting and syntax highlighting to your objects. Here's the same example, but this time formatted using Awesome Print:

```
>> g = Game.find(124)
Game Load (0.1ms) SELECT `games`.* FROM `games`
  WHERE `games`.`id` = 124 LIMIT 1
#<Game:0x007f951a7af8f8> {
  :id => 124,
  :name => "Zaxxon",
  :description => "A vintage arcade game.",
  :created_at => Mon, 21 Jul 2014 15:42:21 UTC +00:00,
  :updated_at => Mon, 21 Jul 2014 15:42:21 UTC +00:00,
  :release_date => 1982,
  :manufacturer_id => 2,
  :slug => "zaxxon"
}
```

Awesome Print is also useful for reviewing a collection of objects:

¹²²https://github.com/michaeldv/awesome_print

```
>> Game.limit(2).to_a
Game Load (0.3ms) SELECT `games`.* FROM `games` LIMIT 2
[
  [0] #<Game:0x007fab4a955ba8> {
    :id => 1,
    :name => "1942",
    :description => "A vintage arcade game.",
    :created_at => Mon, 21 Jul 2014 15:42:20 UTC +00:00,
    :updated_at => Mon, 21 Jul 2014 15:42:20 UTC +00:00,
    :release_date => 1984,
    :manufacturer_id => 1,
    :slug => "1942"
  },
  [1] #<Game:0x007fab4a955770> {
    :id => 2,
    :name => "After Burner II",
    :description => "A vintage arcade game.",
    :created_at => Mon, 21 Jul 2014 15:42:20 UTC +00:00,
    :updated_at => Mon, 21 Jul 2014 15:42:20 UTC +00:00,
    :release_date => 1987,
    :manufacturer_id => 2,
    :slug => "after-burner-ii"
  }
]
```



At the time of this writing Awesome Print seems to be having trouble with Rails 4 arrays, requiring you to explicitly convert the results into an array using `to_a`. No big deal for the moment, and I'd imagine the matter will be resolved soon.

You can easily review a model's attributes simply by referencing the model name, like so:

```
>> Game
class Game < ActiveRecord::Base {
  :id => :integer,
  :name => :string,
  :description => :string,
  :created_at => :datetime,
  :updated_at => :datetime,
  :release_date => :integer,
  :manufacturer_id => :integer,
  :slug => :string
}
```

Plenty of other options exist, including integration with the Rails logger. See the project [README](#)¹²³ for a complete summary of what's available.

To install Awesome Print and integrate it into your irb and Rails consoles, begin by adding the following line to your project Gemfile:

```
gem 'awesome_print', group: :development
```

After updating your project bundle, open up your IRB preferences file `.irbrc`, and add the following two lines:

```
require "awesome_print"
AwesomePrint.irb!
```

Reboot your project's Rails console, and Awesome Print's awesome features will be at your disposal!

Introducing TablePrint

TablePrint creator [Chris Doyle](#)¹²⁴ clued me in to his fine gem not long after the first version of this book published, and I found it so useful that adding this section became an immediate priority. The premise behind TablePrint is simple: tabular output of query results within the Rails console. One example is all you need to understand its utility:

```
>> tp Location.limit 5
Location Load (0.7ms) SELECT `locations`.* FROM `locations` LIMIT 5
ID | NAME | STREET | CITY
---|-----|-----|-----
1 | Ethyl & Tank | 19 13th Avenue | Columbus
2 | Buffalo Wild Wings Hilliard | 1710 Hilliard Rome Road | Hilliard
3 | Plain City Lanes & Pizza | 325 Jefferson Avenue | Plain City
4 | Dave & Buster's Hilliard | 3665 Park Mill Run Dr | Hilliard
5 | Galaxy Games & Golf | 3700 Interchange Rd | Columbus
Printed with config
>>
```

In order to achieve this particular outcome I first configured TablePrint to only show the four selected columns using the `set` method:

¹²³https://github.com/michaeldv/awesome_print

¹²⁴<https://github.com/arches>

```
>> tp.set Location, :id, :name, :street, :city
```

You can later reset the column restriction using the `clear` method:

```
>> tp.clear Location
```

If you don't first identify a select set of columns, TablePrint will by default output all record attributes, or show `Method Missing` for any attributes not selected using the `select` method, therefore in most instances I'd say TablePrint really shines when you want to repeatedly review a particular set of record attributes over the course of a console session. Alternatively, if you'd like to have a quick look at a record alongside a relation, such as each location and its corresponding state name, you can specify the desired attributes like so:

```
>> tp Location.limit(5), 'name', 'state.name'
```

NAME	STATE.NAME
Ethyl & Tank	Ohio
Buffalo Wild Wings Hilliard	Ohio
Plain City Lanes & Pizza	Ohio
Dave & Buster's Hilliard	Ohio
Galaxy Games & Golf	Ohio

To install TablePrint all you need to do is add it to your Gemfile:

```
gem 'table_print', group: :development
```

Update your project bundler and restart the Rails console to begin taking advantage of TablePrint!

The Better Errors Gem

Ruby on Rails does a better job than many other frameworks in regards to presenting exception backtraces within the browser window (when running in development mode), but even so they can be pretty painful to review. Consider for instance even this simple trace which appeared due to a typo in the Active Record `order` statement:

NoMethodError in LocationsController#index

undefined method `orderr' for #<Class:0x007f82ad3e0378>

Extracted source (around line #13):

```
11   def index
12
13     @locations = Location.orderr('name asc').paginate(:page => params[:page], :per_page => 5)
14
15   end
16
```

Rails.root: /Users/wjgilmore/Software/dev.arcadenomad.com

[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)

app/controllers/locations_controller.rb:13:in `index'

Request

Parameters:

None

[Toggle session dump](#)

[Toggle env dump](#)

Response

Headers:

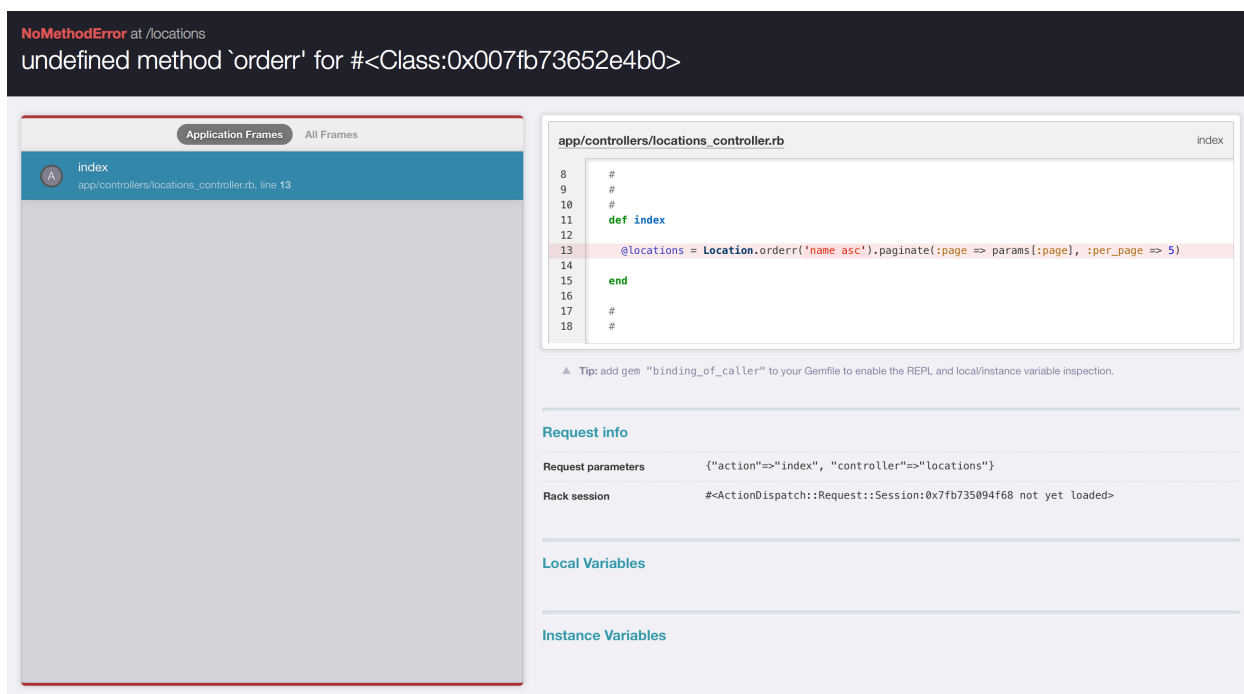
None

Rails' default exception trace is ugly

Although the trace contains everything you need to debug the problem, it's just not particularly pleasant to look at, and the information seems disconnected and in disarray. Because you're going to spend a fair bit of time reviewing stack traces, doesn't it make sense to do so in the most organized and visually appealing way possible? Enter [Better Errors](#)¹²⁵, a fantastic gem created by [Charlie Somerville](#)¹²⁶. Once installed, your stack traces will be magically transformed to look like that presented in the below screenshot.

¹²⁵https://github.com/charliesome/better_errors

¹²⁶<https://github.com/charliesome>



Reviewing traces with Better Errors

A huge visual improvement, to say the least! To install Better Errors all you need to do is add it to your Gemfile:

```
gem 'better_errors', group: :development
```

Update your project bundler and restart the Rails server, and the next time an exception occurs it will be displayed using Better Errors' fantastic user interface!

The Pry Shell

The IRB shell and Rails console are indispensable tools, yet both are missing a few key features that many developers have long clamored for. The [Pry](http://pryrepl.org/)¹²⁷ replacement shell attempts to fill in some of these holes, offering capabilities such as command and navigation shell integration, documentation and source code browsing, and syntax highlighting.

Using Pry in conjunction with your Rails application is easy; just add the following line to your project Gemfile:

```
gem 'pry-rails', group: development
```

After running `bundle install` Pry will automatically replace the default console. Let's login to the Pry (done by running `rails console` per usual), and get acquainted with a few commands:

¹²⁷<http://pryrepl.org/>


```
$ rails console
Loading development environment (Rails 4.1.1)
```

Once logged in, you have access to all of the same capabilities found in the default console, such as retrieving a game record:

```
[1] pry(main)> g = Game.find(112)
Game Load (0.3ms)  SELECT `games`.* FROM `games` WHERE `games`.`id` = 112 LI\
MIT 1
=> #<Game id: 112, name: "Time Pilot '84", description: "A vintage arcade game." \
',
  created_at: "2014-07-21 15:42:21", updated_at: "2014-07-21 15:42:21",
  release_date: 1984, manufacturer_id: 9, slug: "time-pilot-84">
```

Of course, we're interested in what Pry has to offer above and beyond these default capabilities. One useful feature is the ability to easily retrieve a list of model names (and their attributes) using the `show-models` command:

```
[1] pry(main)> show-models
Arcade
  id: integer
  game_id: integer
  location_id: integer
  comment: string
  belongs_to :game
  belongs_to :location
...
State
  id: integer
  name: string
  abbreviation: string
  created_at: datetime
  updated_at: datetime
  locations_count: integer
  has_many :locations
```

If you're only interested in a specific model you can use the `show-model` command:

```
[2] pry(main)> show-model Game
Game
  id: integer
  name: string
  description: string
  created_at: datetime
  updated_at: datetime
  release_date: integer
  manufacturer_id: integer
  slug: string
  belongs_to :manufacturer
  has_many :arcades
  has_many :locations (through :arcades)
```

One of my favorite Pry features is the ability to review documentation, and can parse both [RDoc](https://github.com/rdoc/rdoc)¹²⁸ and [YARD](https://github.com/lsegal/yard)¹²⁹ (ArcadeNomad uses RDoc). For instance to learn more about the `Location` model's `address` method we'll first "change" into the `Location` context and then call the `show-doc` command, passing along the name of the method whose documentation we'd like to review:

```
[3] pry(Location):1> show-doc address
```

```
From: /Users/wjgilmore/Software/dev.arcadenomad.com/app/models/location.rb @ line 91:
```

```
Owner: Location
Visibility: public
Signature: address()
Number of lines: 8
```

Methods

Concatenates the various address-related attributes together into a single string.

return `[String, false]` the concatenated address or false if the errors array is not empty.

Once you're done with the `Location` model you can return to the main context using `exit`:

¹²⁸<https://github.com/rdoc/rdoc>

¹²⁹<https://github.com/lsegal/yard>

```
[4] pry(Location):1> exit
[5] pry(main)>
```

You can also review a project file's source code from within Pry:

```
[5] pry(main)> cd LocationsController
class LocationsController < ApplicationController

  # Retrieve a paginated array of locations, ordering the results
  # by name in ascending fashion.
  def index

    @locations = Location.order('name asc').paginate(:page => params[:page], :per\
r_page => 20)

  end
  ...
end
```

These introductory examples really only scratch the surface of Pry's capabilities, and the reality is this topic deserves an entire chapter unto itself. So be sure to spend some time exploring this replacement console's features (executing `help` from within the Pry console would be a great start); I'd imagine it won't be too long before you wonder how you ever lived without it!

Useful Links

Be sure to check out the following links to learn more about debugging Rails applications:

- “[Debugging Rails Applications](http://guides.rubyonrails.org/debugging_rails_applications.html)”¹³⁰: This section of the Rails manual introduces the debug helper, logger, and debugger gem, among other topics, and concludes with a useful list of third-party debugging tools and additional referential sources.
- “[Debugging Rails Applications in Development](http://nofail.de/2013/10/debugging-rails-applications-in-development/)”¹³¹: Peter Schroeder penned a lengthy survey of various debugging techniques and tools.
- “[Pry \(and Friends\) with Rails](http://www.sitepoint.com/pry-friends-rails/)”¹³²: SitePoint contributor [Benjamin Tan Wei Ho](http://benjaminanweihao.github.io/)¹³³ penned a great introduction to Pry that's well worth reading if the topic is new to you.

¹³⁰http://guides.rubyonrails.org/debugging_rails_applications.html

¹³¹<http://nofail.de/2013/10/debugging-rails-applications-in-development/>

¹³²<http://www.sitepoint.com/pry-friends-rails/>

¹³³<http://benjaminanweihao.github.io/>

Optimizing Your Application

As your project grows in size and ambition, its performance will necessarily suffer. Despite the great computer scientist [Donald Knuth's](http://en.wikipedia.org/wiki/Donald_Knuth)¹³⁴ declaration that ‘ “Premature optimization is the root of all evil”, you’re not going to want to put off resolving potential bottlenecks for too terribly long. In this section I’ll offer a few pointers pertaining to fairly easy steps you can take to dramatically improve the performance of Active Record and your database in general.



Until I can figure out a proper way to organize the information, I’m for the time being going to omit any database-specific optimization advice in this section and instead just focus on a few optimization-related fundamentals that will be of benefit to all readers. Stay tuned as in a forthcoming update I’ll include MySQL- and PostgreSQL-specific sections.

Using Indexes

Ensuring your database is properly indexed is one of the easiest steps you can take towards ensuring your application data can be quickly retrieved. This is because lacking a proper index, a database is forced to sequentially read all rows in a table to find the desired record (or records), a task that can slow dramatically as the tables increase in size. The index eliminates this sequential read requirement by storing a copy of the data in a special data structure such as a [B-tree](http://en.wikipedia.org/wiki/B-tree)¹³⁵ alongside links to the original records. When an indexed column (or columns) are to be consulted via a query, the database will use the index to retrieve the desired records, and then use the link to return the original records, resulting in a much faster response time.

Indeed, the matter of indexing is so crucial to database performance that you can define indexes directly within your project migrations. Further, Rails will automatically index certain columns for you, such as the `id` column, however it will *not* automatically index foreign keys (such as the `locations` table’s `state_id` or `category_id` columns, meaning you’ll have to create those yourself.

When creating a new table, you’ll not want to just blindly index all of the columns, as over-indexing your database can actually *hurt* performance! Instead, ask yourself which columns will play roles in record retrieval, and begin by indexing them. For instance, when the `ArcadeNomadLocation` model was created, the corresponding `locations` table migration specified that the `name` column should be indexed, because it was obvious `ArcadeNomad` users would want to search for locations by name and therefore the `name` column would be consulted in order to retrieve the desired record(s):

¹³⁴http://en.wikipedia.org/wiki/Donald_Knuth

¹³⁵<http://en.wikipedia.org/wiki/B-tree>

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

When the [FriendlyId](#)¹³⁶ gem was later added, the `slug` column was indexed because the slug would be passed along via the URL in order to produce friendlier URLs and therefore used to retrieve the associated location:

```
class AddSlugToLocations < ActiveRecord::Migration
  def change
    add_column :locations, :slug, :string
    add_index :locations, :slug, unique: true
  end
end
```

Suppose you created a search interface that allowed users to filter records on *both* the `locations` table's `state` and `category` columns. Because both will play a role in the particular query used to execute this search, you can create a multi-column index:

```
add_index :locations, [:state_id, :category_id]
```

Useful Links

A great deal has been written about database indexing, so I encourage you to spend some time searching the web for tutorials and other reference material. To get you started, check out these links:

- [Wikipedia's Database Index Entry](#)¹³⁷: Wikipedia offers a pretty solid definition of a database index.
- ["Efficient Use of PostgreSQL Indexes"](#)¹³⁸: The Heroku team has put together a brief but well-written primer to PostgreSQL indexing.

¹³⁶https://github.com/norman/friendly_id

¹³⁷http://en.wikipedia.org/wiki/Database_index

¹³⁸<https://devcenter.heroku.com/articles/postgresql-indexes>

- “[How MySQL Uses Indexes](#)”¹³⁹: MySQL users not intimately familiar with how MySQL uses indexes should definitely take the time to carefully read this section of the manual.
- “[When to Add What Indexes in Rails](#)”¹⁴⁰: This excellent Stack Overflow thread offers some excellent advice regarding indexing as applied to a Rails application.
- “[Efficient Use of PostgreSQL Indexes](#)”¹⁴¹: Heroku has put together a great introduction to PostgreSQL indexing.

Caching

Rails has long offered a [caching feature](#)¹⁴² that never seems to have gotten a lot of press as compared to other features. This is a shame, because Rails’ native caching capabilities are pretty mature, not to mention flexible, allowing you to cache pages, actions, specific parts of a page (known as *fragment caching*), and database result sets. The database result set caching is actually not as useful as you might think, because it only applies to a query executed more than once within the same request, a behavior that isn’t common in a traditional web application. Not to mention with caching enabled, the database result set caching is automatically enabled with no further interaction on your part. So in this section I’ll focus on briefly introducing you to the other three types of caching (page, action, and fragment).



All mainstream databases (MySQL and PostgreSQL included) offer powerful SQL caching features that can dramatically improve query performance. Be sure to consult your chosen database’s manual for more information.

Enabling Caching

Even if you take the necessary steps to cache data using Rails’ native features, keep in mind caching is disabled by default in the development and test environments, and enabled in production (meaning in development/test the caching statements will execute to no effect). However, if you’re not familiar with how Rails’ caching features work, you’ll probably want to spend some time getting familiar with them in your development environment before deploying any such features to production. To enable caching in your development environment open `config/environments/development.rb` file and locate the following line:

```
config.action_controller.perform_caching = false
```

Change this setting to `true` and save the file, restarting your server to enable the change.

¹³⁹<http://dev.mysql.com/doc/refman/5.7/en/mysql-indexes.html>

¹⁴⁰<http://stackoverflow.com/questions/3658859/when-to-add-what-indexes-in-a-table-in-rails>

¹⁴¹<https://devcenter.heroku.com/articles/postgresql-indexes>

¹⁴²http://guides.rubyonrails.org/caching_with_rails.html

Caching Pages

Sites like ArcadeNomad are necessarily database-driven, yet tend to serve pages that rarely change. For instance, is it really necessary to repeatedly query the database every time a user wants to learn more about the [16-bit Bar](http://arcadenomad.com/locations/16-bit-bar)¹⁴³? Probably not, particularly if your trusty administrator is only logging in once daily to review the latest submissions and update other information. Using page caching, you can completely bypass the Rails stack altogether and just serve the cached page back to the user.

If you're using Rails 4.0 or greater, page caching has been removed from the Rails core and made available via a [gem](https://github.com/rails/actionpack-page_caching)¹⁴⁴. Therefore Rails 4.0+ users should install the gem by adding the following line to their Gemfile:

```
gem 'actionpack-page_caching'
```

After saving the changes run `bundle install` to install the gem. If you're still running Rails 3 then this feature is part of the core and no further installation-related actions are necessary (other than enabling caching as described above).

Next, you'll want to identify the location where cached pages are stored. You'll do this by adding the following line to the appropriate environment initialization file:

```
config.action_controller.page_cache_directory = "#{Rails.root.to_s}/public/cache"
```

Finally, you'll identify the pages you'd like to cache. To do so you'll identify the desired actions using the `caches_page` class method. For instance, to cache the `Locations` controller's `index` and `show` actions, you'll pass `index` and `show` to `caches_page` at the top of the `Locations` controller, like so:

```
class LocationsController < ApplicationController

  caches_page :index, :show

end
```

With the changes in place, the next time the `Locations`' controller's `index` or `show` actions are requested, the page will be served up as normally however its contents will additionally be cached to the directory identified by `config.action_controller.page_cache_directory`. There is one issue to resolve though, the page will remain cached until the cached file is removed! You'll only want to do this when the data comprising the page is updated. If it's data accessed via a model, you can call the `expire_page` method in the model's associated `update` method:

¹⁴³<http://arcadenomad.com/locations/16-bit-bar>

¹⁴⁴https://github.com/rails/actionpack-page_caching

```
expire_page :action => show
```

Alternatively, if you edited other data found in the page, such as the view layout or non-model content, you can enter the cache directory and manually delete the cached page. The desired cached page will be easy to find because it will be saved using a path and naming convention identical to that found in the URL. For instance the `http://arcadenomad.com/locations/16-bit-bar.html` page would be cached within `public/cache/locations/16-bit-bar.html`.

Page caching's major drawback is it can't be used in conjunction with any action that executes in conjunction with `before_filter`, because logically the action's behavior is intended to be affected in some way by the outcome of that filter. You can however work around this issue by caching actions, introduced next.

Caching Actions

Action caching resolves the major drawback to page caching in that it allows you to cache actions associated with a `before_filter` class method, allowing you to for instance authenticate a user before serving the cached action. Like page caching, Rails' action caching feature was removed from the Rails core in version 4.0. Rails 4+ users can add action caching capabilities to their application by adding the following line to the project Gemfile:

```
gem 'actionpack-action_caching'
```

After saving the changes run `bundle install` to make the gem available to your application. If you're still running Rails 3 then this feature is part of the core and no further installation-related actions are necessary (other than enabling caching as described above).

Once installed you can identify the actions you'd like to cache using the `caches_action` class method. For instance, suppose you decided to convert ArcadeNomad into a subscription service, thereby allowing access to solely paid subscribers. You'd possibly use a `before_filter` to ensure the user had authenticated prior to serving the action. Further, you wanted to cache the `Locations` controller's `index` and `show` actions. The top of the `Locations` controller might look like this:

```
class LocationsController < ApplicationController

  before_filter :authenticate

  caches_action :index, :show

  ...

end
```


The cached action output will by default be stored in your project's `tmp/cache` directory, and the cache file name corresponds to the route used to access the action. You can modify this naming convention using the `cache_path` option. Additionally, you have more flexibility regarding how and when the action caches are expired as compared to page caching. For instance, you can use the `expires_in` option to identify a specific cache lifetime, such as twelve hours from the time the action cache is generated:

```
caches_action :index, expires_in: 12.hours
```

Alternatively you can force immediate expiration of an action using `expire_action`:

```
def update

  expire_action :action => :index

  ...

end
```

Finally, I'd like to note you can also forego caching of an action's layout by passing `layout: false`:

```
caches_action, expires_in: 12.hours, layout: false
```

This is useful when part of the layout updates according to some sort of criteria such as an authenticated user's username. When `layout` is set to `false`, only the action view will be cached, and the dynamic layout wrapped around it at request time.

Caching Fragments

Fairly simplistic applications can get away with page or action caching, but more complicated applications often consist of pages comprised of multiple different types of data, with each having specific caching requirements (including none at all). Fortunately, even in these sort of advanced situations you can take advantage of caching thanks to *fragment caching*, or caching of just part of the page. Unlike page and action caching you'll configure fragment caching in the view, and additionally unlike these other types of caching, fragment caching remains part of the Rails core, meaning no additional steps are required for installation (other than ensuring caching is enabled as described previously).

To cache just a portion of a Rails view you'll wrap it in a cache block. For instance, it doesn't make a lot of sense to repeatedly calculate the distance between multiple locations within the `Locations` controller's `show` action, so we can cache that fragment for twelve hours before calculating the distances anew:

```
<% cache(expires_in: 12.hours) do %>
  <% Location.near([@location.latitude, @location.longitude], 20).each
    do |nearby_location| %>
    <li class="list-group-item">
      <h4 class="list-group-item-heading">
        <%= link_to nearby_location.name, location_path(nearby_location) %>
      </h4>
      ( <%= nearby_location.distance_from([@location.latitude,
        @location.longitude]).round(2) %> miles)
    </li>
    <% end %>
  <% end %>
```

Like page and action caching, you can also manually expire a cache using the `expire_fragment` method:

```
def update

  expire_fragment(controller: 'locations', action: 'show')

  ...

end
```

Other useful fragment caching options exist, so be sure to check out the [relevant part of the Rails manual](#)¹⁴⁵ for all of the details.

Useful Links

Check out the following links for more information about Rails and caching:

- [Caching with Rails: An Overview](#)¹⁴⁶: The Rails manual offers a straightforward introduction to Rails' caching features.
- [The actionpack-page_caching Gem](#)¹⁴⁷: Rails' page caching feature was moved out of the core and into its own gem beginning with version 4.0. The GitHub README offers guidance regarding how the gem should be used.
- [The actionpack-action_caching Gem](#)¹⁴⁸: Like page caching, Rails' action caching feature was moved out of the core and into its own gem beginning with version 4.0. See the README for more information about how to use the gem.

¹⁴⁵http://guides.rubyonrails.org/caching_with_rails.html#fragment-caching

¹⁴⁶http://guides.rubyonrails.org/caching_with_rails.html

¹⁴⁷https://github.com/rails/actionpack-page_caching

¹⁴⁸https://github.com/rails/actionpack-action_caching

Other Worthy Optimization Solutions

There are *plenty* of other optimization utilities worthy of discussion in this chapter, but alas due to time constraints I'll save expanded coverage for a future update. In the meantime have a look at these following four gems, all of which focus on helping you to identify performance constraints:

- [Bullet](#)¹⁴⁹: The Bullet gem monitors the queries executed by your application and offer advice regarding how certain queries could be optimized.
- [New Relic Ruby Agent](#)¹⁵⁰: The New Relic Ruby Agent gathers diagnostic information about your application and sends it to the New Relic dashboard where you can monitor your application's health in real-time. This is a very slick solution I used on a recent project and no doubt I'll soon be integrating it into ArcadeNomad.
- [Peek](#)¹⁵¹: The Peek gem gathers diagnostic information about your application and presents it in a toolbar that can be embedded anywhere within your application layout. Among the diagnostics you can choose to include in the toolbar are MySQL, Git, garbage collection, PostgreSQL, and others.
- [Rails Footnotes](#)¹⁵²: Like Peek, Rails Footnotes displays a variety of debugging information within your application layout, including information about cookies, queries, the filter chain and sessions.

Useful Links

Check out these interesting links to learn more about optimization:

- [“Improve Rails Performance by Adding a Few Gems”](#)¹⁵³: Marian Posaceanu recently blogged about several utilities and interesting gems that can collectively contribute to a faster Rails application.
- [“Introduction to Ruby Code Optimization”](#)¹⁵⁴: Nicolas Zermati penned a two part blog entry on Ruby code optimization, covering a variety of topics including profiling, caching, benchmarking, and the importance of choosing the proper data structure, among other topics.

Conclusion

All good things must come to an end, including the great deal of fun I've had writing this book. I hope you've found it at least somewhat useful, and invite you to stay tuned as in the coming weeks and months I'll be both improving and expanding the book, making all updates available to existing customers! In the meantime please e-mail me with your questions, comments and suggestions at wj@wjgilmore.com!

¹⁴⁹<https://github.com/flyerhzm/bullet>

¹⁵⁰<https://github.com/newrelic/rpm>

¹⁵¹<https://github.com/peek/peek>

¹⁵²<https://github.com/josevalim/rails-footnotes>

¹⁵³<http://marianposaceanu.com/articles/improve-rails-performance-by-adding-a-few-gems>

¹⁵⁴http://www.synbioz.com/blog/optimize_ruby_code

Chapter 7. Integrating User Accounts with Devise

Providing users with the opportunity to create and manage an account opens up a whole new world of possibilities in terms of enhanced interactivity and the ability for users to create and view custom content. However, there are a great many matters one has to take into consideration in order to integrate account authorization and management, including user registration, secure storage of user credentials, user sign in and sign out, lost password recovery, profile management, and general integration of customizable features into your web application.

Fortunately, there exists an incredibly useful and popular gem called [Devise](#)¹⁵⁵ that can dramatically reduce the amount of work you'd otherwise have to do in order to integrate account authorization and management features. In addition to essentially implementing all of the aforementioned features for you, Devise is packaged with numerous optional capabilities including requirement of account confirmation, account locking after a certain number of failed sign in attempts, tracking of users according to IP address and the number of times they've signed into the application, integration with third-party authentication providers such as Amazon.com, GitHub and Google, and much more.

In this chapter I'll show you how I used Devise to integrate basic user authentication and account management features into ArcadeNomad. I'll also show you how to customize various aspects of Devise, including the registration and sign in forms, and create several custom convenience routes.



While this chapter does guide you through the Devise features most commonly integrated into a Rails application, it is by no means offers an exhaustive introduction to this powerful Gem. You'll almost certainly want to make additional modifications to Devise's default behavior, many of which involve configuration changes that aren't obvious. Without question the best available resource for carrying out these changes is the [Devise wiki](#)¹⁵⁶, so be sure to consult this resource before performing any further online research, and of course feel free to e-mail me if you can't find the answer.

Installing Devise

To install Devise, add the following line to your project `Gemfile`:

¹⁵⁵<https://github.com/plataformatec/devise>

¹⁵⁶<https://github.com/plataformatec/devise/wiki>

```
gem 'devise'
```

After updating the project bundle (execute `bundle install`), run the Devise installer:

```
$ rails generate devise:install
```

This creates two files, including:

- `config/initializers/devise.rb`: This initializer defines a great number of settings that determine among other things the required length of a user password should Devise validation be used, the maximum length of a user session before being asked to login again, and the number of times a user can attempt to login before being locked out of an account. You should take a few moments now to peruse this file in order to understand the many options at your disposal when using this powerful gem.
- `config/locales/devise.en.yml`: The various messages Devise users to communicate with users (“Your account was successfully confirmed”, “Invalid email or password”, etc.) has been translated into more than three dozen languages, among them German, Greek, Italian, Japanese, and Russian. Devise integrates these translations in accordance with the [Rails’ native internationalization framework](#)¹⁵⁷, meaning you can easily update Devise’s messages to reflect the native language, currency and other customs of your particular audience. The file `config/locales/devise.en.yml` contains the English language translations of Devise’s various messages, which is what Devise uses by default.

In addition to creating these two files, this command will output a list of additional configuration requirements such as setting ActionMailer’s default URL, setting `root_url` in the routes file, and configuring the display of flash messages. All of these requirements have already previously been configured in *ArcadeNomad*, however be sure to carefully read through this list in case you need to update your own application accordingly.

Creating the User Account Model

A model and underlying database table is logically used by Devise to house user data and manage the various behaviors associated with user accounts. You can easily create a Devise-enhanced user model like so:

¹⁵⁷<http://guides.rubyonrails.org/i18n.html>

```
$ rails generate devise User
  invoke  active_record
  create   db/migrate/20140908154949_devise_create_users.rb
  create   app/models/user.rb
  invoke   rspec
  create    spec/models/user_spec.rb
  invoke    factory_girl
  create     spec/factories/users.rb
  insert    app/models/user.rb
  route    devise_for :users
```

You're free to use any model name you'd like for user management, however `User` is fairly standard.

Following the creation of a new model you would typically run the corresponding migration in order to create the companion table. However in this case it is important that you not do so just yet! This is because the newly created model can be further enhanced with several Devise-specific features, and these features require the addition of their own columns in the `users` table. Therefore you'll want to be sure you've defined these features to suit your particular needs before running the migration. Open the `User` model and you should see the following code:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

Devise has already enhanced the model with several default options, including:

- `database_authenticatable`: Use the database both for storing the user password (encrypted, of course) and for validating the user when attempting to sign into the web site.
- `registerable`: Allow users to create a new account by registering.
- `recoverable`: Allow the user password to be reset by sending instructions to the user's registered e-mail address.
- `rememberable`: Allow the user's session to be managed in a session cookie, affording the user the convenience of not being forced to repeatedly sign into the web site upon each subsequent return.
- `trackable`: Track the user's IP address and the number of times the user has logged into the web site.
- `validatable`: Include default validation of the user's e-mail address and password.

Additionally, several other options are available but aren't enabled by default. These include:

- `confirmable`: Require the user to confirm a new registration by clicking on a one-time URL sent via e-mail.
- `lockable`: Lock the user's account for a specified period of time after multiple failed sign in attempts.
- `timeoutable`: Automatically log the user out after a specified period of inactive time.
- `omniauthable`: Add [Omniauth](https://github.com/intridea/omniauth)¹⁵⁸ support, allowing users to register and sign in using third-party credentials. Unlike many of the other options listed here, additional steps are required to implement this feature which are currently out of the scope of this chapter.

ArcadeNomad uses the default set of options enabled within the `User` model, however you should at this time update this option list in order to accommodate the needs of your specific application. Once you are satisfied, save the changes and create the corresponding `users` table by running the migration:

```
$ rake db:migrate
== 20140908154949 DeviseCreateUsers: migrating =====
-- create_table(:users)
  -> 0.0177s
-- add_index(:users, :email, {:unique=>true})
  -> 0.0101s
-- add_index(:users, :reset_password_token, {:unique=>true})
  -> 0.0103s
== 20140908154949 DeviseCreateUsers: migrated (0.0382s) =====
```

With this complete, we're ready to begin integrating account registration into the application!

Adding User Registration Capabilities

Believe it or not, after generating the `User` model there's actually very little left to do in most cases because the mere act of adding Devise to your application goes a long way towards integrating the various account-related features into your application. If you return to the feedback generated by the rails `generate devise User` command you'll see the following line has been added to the `config/routes.rb` file:

```
devise_for :users
```

This is a helper that defines numerous new application routes, including `/users/sign_up`. If in your development environment you navigate to `http://0.0.0.0:3000/users/sign_up` you'll see a simple registration form asking the user to supply an e-mail address and password. Complete

¹⁵⁸<https://github.com/intridea/omniauth>

this form by providing a valid e-mail address and a password consisting of between eight and 128 characters, submit the form and Devise will create the new account, redirect you to the home page, and display the message “Welcome! You have signed up successfully.” (in your application’s flash notice). The users table will be updated to include the newly registered user’s record. , Here is an example of what the table looks like when the default features are used, keeping in mind your table structure and contents will vary according to the particular set of Devise options you enabled in the User model:

```
mysql> select * from users\G
***** 1. row *****
      id: 1
    email: wj@wjgilmore.com
 encrypted_password: $2a$10$RLE1tQ65XYDmw9TpbOot.eq0/oNdk6VuiaEiojHxWW/Q4jOri\
coi2
 reset_password_token: NULL
reset_password_sent_at: NULL
  remember_created_at: NULL
    sign_in_count: 1
 current_sign_in_at: 2014-09-08 16:28:42
  last_sign_in_at: 2014-09-08 16:28:42
 current_sign_in_ip: 127.0.0.1
  last_sign_in_ip: 127.0.0.1
      created_at: 2014-09-08 16:28:42
     updated_at: 2014-09-08 16:28:42
1 row in set (0.00 sec)
```

Notice the password is encrypted. Devise uses a hashing algorithm known as [bcrypt](http://en.wikipedia.org/wiki/Bcrypt)¹⁵⁹ to ensure passwords can’t be decrypted even if the database were to be somehow compromised and an attacker gained access to the encrypted passwords.

If you did not identify the User model as `confirmable` then Devise will automatically sign the user into the account following registration. Of course, this won’t be readily apparent to the user after being redirected to the home page, so you’ll need to make a change to your project’s layout in order to let the user know whether he’s currently logged in. Open `app/views/layouts/application.html.erb` and add the following code at a convenient location:

¹⁵⁹<http://en.wikipedia.org/wiki/Bcrypt>


```
<% if user_signed_in? %>
  <%= link_to "Hi, #{current_user.email}", edit_user_registration_path %>
  <%= link_to 'Sign out', destroy_user_session_path, :method => :delete %>
<% else %>
  <%= link_to 'Sign in', new_user_session_path %> |
  <%= link_to 'Create account', new_user_registration_path %>
<% end %>
```

Devise’s `user_signed_in` helper is used to determine whether the user is currently signed into the application. If so, a second helper (`current_user`) is used to retrieve the signed in user’s model, and in the case of this example a message like `Hi, wj@wjgilmore.com` is displayed and linked to the user’s account management page (the route of which can be retrieved using the `edit_user_registration_path` path helper (I’ll discuss the account management page later in the chapter):

```
<%= link_to "Hi, #{current_user.email}", edit_user_registration_path %>
```

Additionally, a link is provided directing signed in users to the sign out path.

```
<%= link_to 'Sign out', destroy_user_session_path, :method => :delete %>
```

If the user isn’t logged in, a link to the user sign in form is provided:

```
<%= link_to 'Sign in', new_user_session_path %>
```

Of course, because presumably not all visitors have accounts, a link to the user registration form is also provided:

```
<%= link_to 'Create account', new_user_registration_path %>
```

Next, let’s talk more about the sign in and sign out features.

Adding User Sign In and Sign Out Capabilities

As with registration, Devise automatically integrates user sign in and sign out features, meaning you have little additional work to do in order to begin taking advantage of these features. Navigate to `http://0.0.0.0:3000/users/sign_in` to see the user sign in form, consisting of fields for the e-mail address and password. Presuming you created an account in the last section, and subsequently signed out, give the sign in form a try now. If you left the default Devise option `rememberable` enabled in the `User` model, notice the sign in form also includes a “Remember me” option. After submitting the form you’ll be redirected to the home page and the flash message “Signed in successfully.” is displayed.

If you updated your project’s layout file as described in the previous section, you should see the account sign out link. Recall the `link_to` helper looks like this:

```
<%= link_to 'Sign out', destroy_user_session_path, :method => :delete %>
```

You need to supply the `:method => :delete` option because Devise configures this route to use the DELETE rather than the GET method. If you do not supply the option, the `link_to` helper will create a standard hyperlink which will by default use the GET method, resulting in a 404 when followed. While using the DELETE method to destroy a resource is indeed preferred in terms of adhering to the REST principles first introduced in Chapter 5, it does come at a potential inconvenience: Rails implements HTTP DELETE methods using JavaScript, and specifically relies upon the [jQuery UJS](#)¹⁶⁰ adapter to do so, you'll need to ensure your `app/assets/javascripts/application.js` file includes the following two lines:

```
//= require jquery  
//= require jquery_ujs
```

If including jQuery and the jQuery UJS libraries presents a problem, you can override Devise's default behavior in regards to use of the DELETE HTTP method for this purpose and instead use the GET request in conjunction with sign out. To do so, open `config/initializers/devise.rb` and locate the following line:

```
config.sign_out_via = :delete
```

Change the line to read:

```
config.sign_out_via = :get
```

After saving the changes and restarting the server you'll be able to reference `destroy_user_session_path` without the method option:

```
<%= link_to 'Sign out', destroy_user_session_path %>
```

If you'd like more perspective on Devise's default sign out implementation and the role jQuery UJS plays, be sure to check out [this Stack Overflow discussion](#)¹⁶¹ and [this great thoughtbot blog post](#) titled "A Tour of Rails' jQuery UJS"¹⁶², respectively.

Adding User Account Management Capabilities

In the earlier modifications to the project layout I referenced the `edit_user_registration_path` path helper:

¹⁶⁰<https://github.com/rails/jquery-ujs>

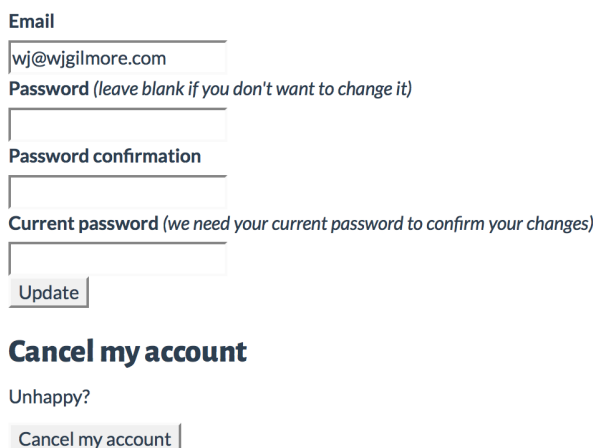
¹⁶¹<http://stackoverflow.com/questions/6557311/no-route-matches-users-sign-out-devise-rails-3>

¹⁶²<http://robots.thoughtbot.com/a-tour-of-rails-jquery-ujs>

```
<%= link_to "Hi, #{current_user.email}", edit_user_registration_path %>
```

Clicking on this link directs users to `/users/edit` where the user can update the account e-mail address, change the account password, and cancel the account (see below screenshot). Chances are you won't want users to be able to cancel their account so easily; if not see [this Devise wiki page](#)¹⁶³ regarding how the routes can be redefined to disallow account cancellation.

Edit User



Email
wj@wjgilmore.com

Password (leave blank if you don't want to change it)

Password confirmation

Current password (we need your current password to confirm your changes)

Update

Cancel my account

Unhappy?

Cancel my account

The default account management form

Chances are you'll eventually want to accumulate other information about your users, such as their name or city-of-residence. I'll show you how to do this in the later section, "Adding Attributes to the User Profile".

Adding Password Recovery Capabilities

If there is anything guaranteed in the universe, it's that users will forget their passwords with great frequency. Therefore providing an automated solution for password recovery is necessary so as to avoid the hassle of dealing with what would eventually become an endless stream of customer support requests. Devise automatically integrates a recovery mechanism, accessible via `/users/password/new`. The user can supply the account e-mail address and instructions for resetting the password will sent via e-mail. I'll show you how to modify the text found in this e-mail in the later section, "Overriding the Devise Views".

You'll want to configure the Devise initializer's `config.mailer_sender` option to reflect the e-mail address used within your Action Mailer configuration (see Chapter 5 for more information about ActionMailer). This option is found in `config/initializers/devise.rb`, and by default looks like this:

¹⁶³<https://github.com/plataformatec/devise/wiki/How-To:-Disable-user-from-destroying-his-account>

```
config.mailer_sender = 'please-change-me-at-config-initializers-devise@example.com'
```

When the user submits an account e-mail address, a one-time URL will be generated and embedded into a password recovery e-mail and sent to the user. Once the user clicks the link he will be directed to a form prompting to create and confirm a new account password.

Restricting Access to Authenticated Users

Restricting access to a particular controller is incredibly easy, requiring only the use of a `before_filter`. For instance suppose ArcadeNomad partnered with various pizzerias and offered printable coupons to registered users. You could restrict access to the coupons controller like so:

```
class CouponsController < ApplicationController

  before_filter :authenticate_user!

  def index
  end

  ...

end
```

If the user is not authenticated, he'll be redirected to the sign in page.

Commonplace Devise Customizations

There are seemingly countless ways in which you can customize Devise to your liking, many of which are documented in [the Devise wiki](https://github.com/plataformatec/devise/wiki)¹⁶⁴. In this section I'll talk about two customizations that seem to be of the highest priority for most developers, showing you how to modify the default Devise views to meet the design standards of your application, and how to add profile attributes to your `User` model beyond the default e-mail address and password.

Overriding the Devise Views

The default Devise forms are functional but probably don't meet the visual standards of your particular project. You can export the various views and e-mail templates used by Devise into files so you can update them to suit your needs. To do so, execute the following command from your project's root directory:

¹⁶⁴<https://github.com/plataformatec/devise/wiki>

```
$ rails g devise:views
  invoke  Devise::Generators::SharedViewsGenerator
  create   app/views/devise/shared
  create   app/views/devise/shared/_links.erb
  invoke  form_for
  create   app/views/devise/confirmations
  create   app/views/devise/confirmations/new.html.erb
  create   app/views/devise/passwords
  create   app/views/devise/passwords/edit.html.erb
  create   app/views/devise/passwords/new.html.erb
  create   app/views/devise/registrations
  create   app/views/devise/registrations/edit.html.erb
  create   app/views/devise/registrations/new.html.erb
  create   app/views/devise/sessions
  create   app/views/devise/sessions/new.html.erb
  create   app/views/devise/unlocks
  create   app/views/devise/unlocks/new.html.erb
  invoke  erb
  create   app/views/devise/mailer
  create   app/views/devise/mailer/confirmation_instructions.html.erb
  create   app/views/devise/mailer/reset_password_instructions.html.erb
  create   app/views/devise/mailer/unlock_instructions.html.erb
```

As you can see quite a few views have been exported, in addition to several e-mail templates. All you need to do at this point is open each view in your IDE and update them accordingly. The sky is the limit in terms of the degree to which you can modify the forms and other visual cues, because there's nothing particularly special about the exported markup. For instance if you navigate to http://arcadenomad.com/users/sign_in¹⁶⁵ you'll see I've updated the sign in form to use Bootstrap styling.

Adding Attributes to the User Profile

The default account management form depicted in the screenshot found in the section “Adding User Account Management Capabilities” is pretty sparse, because the default registration form only asked for two items: an e-mail address and password. Chances are you will want to ask for additional identifying data, such as the user's name, city, or date-of-birth. You can however add these columns pretty easily. Let's work through an example in which we'll add the user's name to the users table and account management form. Begin by creating the necessary migration:

¹⁶⁵http://arcadenomad.com/users/sign_in

```
$ rails g migration addNameToUsers name:string
  invoke  active_record
  create   db/migrate/20140909130956_add_name_to_users.rb
```

Next, incorporate the name column into the users table by running the migration:

```
$ rake db:migrate
== 20140909130435 AddNameToUsers: migrating =====
-- add_column(:users, :name, :string)
   -> 0.0249s
== 20140909130435 AddNameToUsers: migrated (0.0250s) =====
```

Next, open the app/views/devise/registrations/edit.html.erb view and add the appropriate field just as you did in Chapter 5:

```
<div class="form-group">
  <%= f.label :name %>
  <%= f.text_field :name, class: 'form-control' %>
</div>
```

If you reload /users/edit you'll see the new field. However, if you enter a name, submit the form, and subsequently return to /users/edit, you'll see the name has not been persisted. This is because Devise does not for security reasons simply allow updates to any attribute other than those it specifically declares as being safe. There are a number of different solutions to informing Devise of your wish to update another attribute, however in my experience the easiest approach is to add the following code to your project's /app/controllers/application_controller.rb:

```
before_filter :configure_devise_parameters, if: :devise_controller?

protected
def configure_devise_parameters
  devise_parameter_sanitizer.for(:account_update) << :name
end
```

This tells Rails to execute the method `configure_devise_parameters` if the executing controller is Devise-related. In doing so, it will add the name attribute to the Devise parameter sanitizer when updating an account.

Conclusion

Users have grown to expect the opportunity to customize a web application by way of account-driven preferences. Fortunately, the hyperactive ecosystem of Ruby developers have come through once again with the fantastic Devise gem, greatly reducing the amount of work you'd otherwise have to do to implement even the basic account-related features such as registration and sign in. In a future book update I'll be sure to expand upon the material presented in this chapter, in particular talking about Devise's ability to integrate with [Omniauth](https://github.com/intridea/omniauth)¹⁶⁶.

¹⁶⁶<https://github.com/intridea/omniauth>

Chapter 8. Powerful Active Record Plugins

Ruby on Rails is such a powerful solution thanks in large part to its enormous ecosystem. You're probably familiar with prominent gems such as Devise, RSpec, and FriendlyId, but there are also much more focused gems capable of extending specific parts of Rails, including templating (such as [Redcarpet](#)¹⁶⁷), JavaScript ([jquery-ujs](#)¹⁶⁸ for instance), and e-mail (see the very popular [Mail](#)¹⁶⁹ gem). So too can you extend Active Record in a variety of interesting and useful ways, and in this chapter I'll introduce you to a few of my favorite extensions. Keep in mind the gems presented in this chapter are but a sample of the available Active Record gems. For a glimpse into what else is available check out my regularly updated [GitHub gist](#)¹⁷⁰ on the topic.

Tagging Your Content Using the Acts As Taggable On Gem

Over time, traditional [taxonomies](#)¹⁷¹ have been gradually replaced by [metadata](#)¹⁷²-driven approaches to web content categorization on the basis that structured categorization hierarchies are often too inflexible, hindering users' ability to efficiently find content due to a limited selection vocabulary. A metadata-based approach removes such restrictions because information can be classified under a much larger and more descriptive set of keywords (often referred to as *tags*) that more holistically define what a user might expect to take away as a result of accessing that content.

To really understand the difference between these two approaches, consider how [ArcadeNomad](#)¹⁷³ might categorize the arcade games using these two approaches. Using the traditional taxonomical approach the game Asteroids might only be found by navigating through a series of hierarchical categories such as Space and then Shooter whereas using the flat, tag-based approach a much more rich set of classification keywords might give users the ability to immediately locate Asteroids using one of many possible descriptors (see the below figure).

¹⁶⁷<https://github.com/vmg/redcarpet>

¹⁶⁸<https://github.com/rails/jquery-ujs>

¹⁶⁹<https://github.com/mikel/mail>

¹⁷⁰<https://gist.github.com/wjgilmore/9846234>

¹⁷¹http://en.wikipedia.org/wiki/Taxonomy_%28general%29

¹⁷²http://en.wikipedia.org/wiki/Tag_%28metadata%29

¹⁷³<http://arcadenomad.com/>

Tag Cloud

Arcade Bricks **Classic** Driving Flight
Puzzle Sequel **Shooter** Space Tanks World
War II

An example tag cloud

Given this tag cloud, the user could click on the tag that makes the most sense, with logical candidates being Classic, Space, and Shooter.

Integrating content tagging into your Rails application is surprisingly easy thanks to a fantastic gem called [Acts as Taggable On](https://github.com/mbleigh/acts-as-taggable-on)¹⁷⁴. Created by [Michael Bleigh](https://github.com/mbleigh)¹⁷⁵ and maintained by [Joost Baaij](https://github.com/joostbaaij)¹⁷⁶, Acts as Taggable On is one of the latest incarnations of a line of powerful tagging gems long available to the Rails community. In this section I'll introduce you to Acts as Taggable On, explaining how I integrated this simple tagging feature into ArcadeNomad.

Installing the Acts as Taggable On Gem

Install the `acts_as_taggable_on` gem by adding the following line to your project's Gemfile:

```
gem 'acts-as-taggable-on'
```

Save the Gemfile and complete the installation process by executing bundle:

```
$ bundle
...
Installing acts-as-taggable-on (3.4.2)
...
```

After `acts_as_taggable_on` has been installed, you'll need to generate the tables used to store the tags and associations. This is accomplished using a Rake task bundled with the gem:

```
$ rake acts_as_taggable_on_engine:install:migrations
Copied migration 20141021024747_acts_as_taggable_...
Copied migration 20141021024748_add_missing_unique...
```

After the migration files have been created, migrate the database:

¹⁷⁴<https://github.com/mbleigh/acts-as-taggable-on>

¹⁷⁵<https://github.com/mbleigh>

¹⁷⁶<https://github.com/tilsamans>

```

$ rake db:migrate

== ActsAsTaggableOnMigration: migrating =====
-- create_table(:tags)
  -> 0.0319s
-- create_table(:taggings)
  -> 0.0138s
-- add_index(:taggings, :tag_id)
  -> 0.0143s
-- add_index(:taggings, [:taggable_id, :taggable_type, :context])
  -> 0.0116s
== ActsAsTaggableOnMigration: migrated (0.0719s) ===

== AddMissingUniqueIndices: migrating =====
-- add_index(:tags, :name, {:unique=>true})
  -> 0.0111s
-- remove_index(:taggings, :tag_id)
  -> 0.0096s
-- remove_index(:taggings, [:taggable_id, :taggable_type, :context])
  -> 0.0062s
-- add_index(:taggings, [:tag_id, :taggable_id, :taggable_type,
  :context, :tagger_id, :tagger_type],
  {:unique=>true, :name=>"taggings_idx"})
  -> 0.0107s
== AddMissingUniqueIndices: migrated (0.0380s) =====

```

At this point two new tables have been created: `tags` and `taggings`. The `tags` table manages the actual tags, such as `Space` and `Classic`. The `taggings` table is a bit more complex. It acts as the bridge between the model you've identified as *taggable*, and the tags table containing the tags associated with the taggable model's records. Additionally, it defines the *context* of the tag. This is important because you might optionally wish to tag a model using multiple different contexts. For instance, if we were to extend the `Location` model to be taggable, the it might be tagged using two different contexts: `foods` and `neighborhood`. Therefore the `16-Bit Bar & Arcade` in `Columbus, Ohio` might be tagged as `Bar` (using the `foods` tag context) and `Downtown` (using the `neighborhood` tag context). This flexible approach gives you the power to create interfaces that allow users to find data using multiple different tag sets. For instance, users might wish to view only a list of arcades offering pizza or popcorn, or restrict the listing to arcades found on the lower east side.

Keep in mind you aren't required to define taggings in conjunction with a specific context; it's possible to configure a model as merely taggable and then throw all of your tags into a single pot. I'll show you how this is done in the next section.

Configuring a Taggable Model

Let's put the concepts introduced above into action by configuring ArcadeNomad's Game model to be taggable. We'll start with the simple use case first, which is to simply configure the model as capable of being associated with tags:

```
class Game < ActiveRecord::Base

  acts_as_taggable

end
```

Save the file and then open up the Rails console sandbox to test out the tagging capabilities:

```
$ rails console --sandbox
>> location = Game.find_by_name('Asteroids')
Game Load (0.6ms)  SELECT `games`.* FROM `games`
  WHERE `games`.`name` = 'Asteroids' LIMIT 1
#<Game:0x007ffe10b19840> {
  :id => 7,
  :name => "Asteroids",
  :description => "A vintage arcade game.",
  :created_at => Tue, 21 Oct 2014 16:01:22 UTC +00:00,
  :updated_at => Tue, 21 Oct 2014 16:01:22 UTC +00:00,
  :release_date => 1979,
  :manufacturer_id => 5,
  :slug => "asteroids"
}
>> game.tag_list.add('Space')
ActsAsTaggableOn::Tag Load (1.1ms)  SELECT `tags`.* FROM `tags`
  INNER JOIN `taggings` ON `tags`.`id` = `taggings`.`tag_id`
  WHERE `taggings`.`taggable_id` = 7 AND `taggings`.`taggable_type`
    = 'Game' AND (taggings.context = 'tags' AND taggings.tagger_id IS NULL)
=> ["Space"]
>> game.tag_list.count
=> 1
>> game.tag_list.add('Classic')
...
>> game.tag_list
[
  [0] "Space",
  [1] "Classic"
]
```

Obviously you'll want to use other means for adding tags to your application content, such as a form or seeder file. However the general syntax remains the same; just use the `tag_list` collection's `add` method to add new tags, and use the `remove` method to remove tags (not demonstrated here but discussed in the project [README](#)¹⁷⁷).

Creating a Tag Cloud

Users find tag clouds appealing because of the immediate visual payoff; at a glance it is not only apparent what topics are covered on your site, but also thanks to the varied font sizes and weightings, which tags are used with more frequency than others. At the beginning of this section I presented an example of a tag cloud used in the ArcadeNomad application. In this section I'll show you how that cloud was created. Begin by adding the following code to the view helper file associated with the view in which you'd like the tag cloud to appear (found in `app/helpers`):

```
module GamesHelper
  include ActsAsTaggableOn::TagsHelper
end
```

The `TagsHelper` gives you access to the `Acts As Taggable On` `tag_cloud` method, which we'll use in the view to create the tag cloud. Next, in the desired controller action you'll execute the `tag_counts_on` method in conjunction with the desired model within action associated with the view where you'd like the tag cloud to appear. In this example I'd like to create a tag cloud based on the tags associated with the `Game` model:

```
@tags = Game.tag_counts_on(:tags)
```

Next, open the view in which you'd like the tag cloud to appear.

```
<% tag_cloud(@tags, %w(css1 css2 css3 css4)) do |tag, css_class| %>
  <%= link_to tag.name, tag_path(tag), :class => css_class %>
<% end %>
```

Of course, we haven't yet created the `Tag` controller yet therefore if you try to load the view containing this code you'll receive an error. Let's fix this problem next.

Retrieving Games According to Tag

When the tag cloud renders, each tag will be linked to a URL that looks like this: <http://arcadenomad.com/tag/5>¹⁷⁸. Therefore to retrieve a list of games associated with a given tag you'll need to create a `Tag` controller (obviously you can change the controller to whatever you please):

¹⁷⁷<https://github.com/mbleigh/acts-as-taggable-on>

¹⁷⁸<http://arcadenomad.com/tag/5>

```
$ rails g controller Tag show
```

This will generate a controller named `Tag` containing just a single action named `show`. Although only a single action is used I'll go ahead and make this controller RESTful by adding the following line to your `config/routes.rb` file:

```
resources :tag
```

If you do make the controller RESTful you'll want to remove the line `get :tag/show` from `routes.rb`, which was added when the controller was created.

Next, open the `Tag` controller (`app/controllers/tag_controller.rb`) and update the `show` action to look like this:

```
def show
  @tag = Tag.find(params[:id])
  @games = Game.tagged_with(@tag.name)
end
```

This code looks pretty standard, however it's not going to work as expected just yet for one simple reason: the `Tag` model doesn't exist! A `tags` table was generated by `Acts As Taggable On`, however a corresponding model was not. To create it, you'll invoke the model generator per usual, passing along `--migration false` because we don't want to create a migration since the table already exists:

```
$ rails g model Tag --migration false
  invoke  active_record
  create   app/models/tag.rb
  invoke  rspec
  create   spec/models/tag_spec.rb
  invoke  factory_girl
  create   spec/factories/tags.rb
```

With the model created, we can wrap up the feature by adding the following code to the `Tag` controller's `show` view:

```

<h1>Games Tagged as <%= @tag.name %></h1>

<% if @games.count > 0 %>

  <ul>

    <% @games.each do |game| %>

      <li><%= link_to game.name, game_path(game) %></li>

    <% end %>

  </ul>

<% else %>

  <p>This tag is not associated with any games.</p>

<% end %>

```

Styling the Tag Cloud CSS

If you load the view after having created a few tags and tag mappings, you'll see no differentiation between the styling of each tag, regardless of their respective usage frequency. This is because you need to stylize the `css1`, `css2`, `css3`, and `css4` tags injected into the `link_to` method. Add the following four styles to your project's CSS:

```

.css1 { font-size: 1.0em; }
.css2 { font-size: 1.2em; }
.css3 { font-size: 1.4em; }
.css4 { font-size: 1.6em; }

```

Cleaning Up Your Controllers with the Decent Exposure Gem

One of the great advantages of working with various clients is the opportunity to learn from different implementational approaches. Recently I worked with a particularly well-architected Rails application, one which uses several gems that had been unfamiliar prior to this project. One gem in particular really captured my attention, as it can dramatically reduce the amount of boilerplate code

you'd otherwise have to write when building RESTful controllers. That gem is [decent_exposure](#)¹⁷⁹ and it was created by the Rails experts at [Hashrocket](#)¹⁸⁰.

RESTful controllers are great because they follow a conventional format that is immediately recognizable by any Rails developer familiar with the concept, meaning less time is spent deciphering fellow developers' intent and naming conventions. For instance, a RESTful show action is pretty much guaranteed to look like this:

```
class GameController < ApplicationController

  ...

  def show
    @game = Game.find(params[:id])
    respond_with(@game)
  end

  ...

end
```

The show action is then typically accessed via a URL like `http://arcadenomad.com/games/12`. The routing convention dictates that the parameter (in this case, 12) is passed into the show method, which can then be used to query the database for a record associated with that primary key. The decent_exposure developers argue that because everybody knows the find method is going to be used for this purpose, shouldn't we just take that for granted to and automate the call altogether? This means that in the simplest case the show method *doesn't even have to appear in your RESTful controller*. Read that last sentence a few times to let it sink in. The reduction in code isn't limited to show; in fact given a standard RESTful controller enhanced with decent_exposure, the only actions you'll even have to bother including are create and update, and even those are pretty devoid of code compared to what you're probably used to seeing.

The decent_exposure developers seek to resolve another issue: eliminating the exposure of instance variables within your views on the basis that doing so breaks encapsulation. This means when using decent_exposure-enhanced controllers you would no longer reference @game within your show action's corresponding view, but instead just reference game, which is a method exposed by decent_exposure to the view that provides the view with access to the corresponding model's various attributes, methods, and other features.

Installing decent_exposure

Install decent_exposure by adding the following line to your project Gemfile:

¹⁷⁹https://github.com/hashrocket/decent_exposure

¹⁸⁰<https://github.com/hashrocket>

```
gem 'decent_exposure'
```

Save the changes and run `bundle` to install the gem.

Adding decent_exposure to Your Controllers

With the gem installed, you can quickly get started updating your controllers to use `decent_exposure`. I'll revise the aforementioned `Games` controller to use `decent_exposure`, thereby giving you a well-rounded understanding of what's possible. Begin by exposing the `Game` model at the top of the controller:

```
class GamesController < ApplicationController

  respond_to(:html)
  expose(:game)

end
```

With `Game` exposed, `decent_exposure` now knows to begin working its magic in response to various action requests. Here's what the complete revised `Games` controller looks like:

```
class GamesController < ApplicationController

  respond_to(:html)
  expose(:game)

  def create
    game.save
    respond_with(game)
  end

  def update
    game.save
    respond_with(game)
  end

end
```

I know it's difficult to believe I didn't forget to include some code, but that it really the complete `decent_exposure`-enhanced controller! All that remains is to revise your views to use the exposed methods rather than instance variables (again, just remove the `@` from the instance variables), and you're done.

Further Reading

There's no doubt that some of what `decent_exposure` does seems like pure magic but I can assure you it is very well done and in my opinion a required gem when your application uses conventional RESTful controllers. In order to gain a more well-rounded understanding of all that `decent_exposure` has to offer, check out the following resources:

- [decent_exposure README](#)¹⁸¹: This is the place to begin in order to understand the gem's basic features.
- <http://decentexposure.info>¹⁸²: This is the `decent_exposure` gem's companion website, and it contains several lengthy pages explaining the gem's various fundamental and advanced features.

Geocoding Your Models

Sites like ArcadeNomad would be a lot less interesting without the features that can help users find classic arcade games close to their current location. These location-based features are possible determining which arcades are situated within a certain radius of the user's current latitudinal and longitudinal coordinates. As you might imagine, implementing such a feature is fairly complicated, involving several steps:

- Whenever a new arcade is inserted into the database, geocode the address and store the coordinates alongside other arcade-related attributes.
- When a user visits the site, determine the user's latitudinal and longitudinal coordinates based on the IP address.
- Use the [Haversine formula](#)¹⁸³ to identify the arcades residing within a certain radius of the user.

All three of these steps are pretty involved, and not something I'd want to implement alone. Fortunately we can use a great gem and one of several third-party services to handle much of the heavy lifting for us. The gem is called [Geocoder](#)¹⁸⁴, and it was created by [Alex Reisner](#)¹⁸⁵. In this section I'll show you how to integrate Geocoder into your Rails project and use it to plug into the [Google Geocoding](#)¹⁸⁶ and [FreeGeoIP](#)¹⁸⁷ to geocode street addresses and IP addresses, respectively.

Installing the Geocoder Gem

To install the Geocoder gem add the following line to your project's `Gemfile`:

¹⁸¹https://github.com/hashrocket/decent_exposure

¹⁸²<http://decentexposure.info>

¹⁸³http://en.wikipedia.org/wiki/Haversine_formula

¹⁸⁴<https://github.com/alexreisner/geocoder>

¹⁸⁵<https://github.com/alexreisner/>

¹⁸⁶<https://developers.google.com/maps/documentation/geocoding/>

¹⁸⁷<http://freegeoip.net/>

```
gem 'geocoder'
```

Save the changes and run `bundle install` to install the gem.

Geocoding Addresses

With the Geocoder gem installed we can immediately begin using it to convert street addresses into latitudinal and longitudinal coordinates. Geocoder uses the Google Geocoding API by default however it's possible to instead use a number of other competing services, among them [Yahoo BOSS](https://developer.yahoo.com/boss/geo/)¹⁸⁸ and [MapQuest](http://developer.mapquest.com/web/products/open)¹⁸⁹. See the [Geocoder README](https://github.com/alexreisner/geocoder)¹⁹⁰ for more details.

To associate latitudinal and longitudinal coordinates with a model schema you'll need to generate a new migration, adding two `float` columns to the desired table. In the following example I'm adding `latitude` and `longitude` fields to the `locations` table:

```
$ rails g migration AddLatitudeAndLongitudeToLocations \
> latitude:float longitude:float
```

After creating the migration you can add the fields to the table:

```
$ rake db:migrate
```

Keep in mind Geocoder expects the fields to be named `latitude` and `longitude`. If for some reason you need to use different naming conventions you can override them when calling the `geocoded_by` method within your model (more on this method in a moment). For instance the following example overrides `latitude` with `lat` and `longitude` with `lng`:

```
geocoded_by :address, :latitude => :lat, :longitude => :lng
```

Next you need to inform the model as to which fields should be supplied to the geocoder in order to determine the coordinates. For instance you might only wish to geocode according to city and state, but others might wish to use a complete street address (street, city, state and zip code) in order to obtain more accurate results. For `ArcadeNomad` I use the following approach:

¹⁸⁸<https://developer.yahoo.com/boss/geo/>

¹⁸⁹<http://developer.mapquest.com/web/products/open>

¹⁹⁰<https://github.com/alexreisner/geocoder>

```

class Location < ActiveRecord::Base

  ...

  geocoded_by :address

  def address
    [street, city, state.name].compact.join(' ', ' ')
  end

end

```

This approach will gather the street, city, and associated state's name fields, returning them in the format "1234 Jump Street, Dublin, Ohio". Again you're not required to manage all of these address-related components, and can just return the street and zip code for instance. Once these model changes are saved, the supplied address will automatically be geocoded every time a `Location` record is saved or updated. Of course, you can avoid unnecessary API requests by updating your model to only geocode if the address-related components are present or have changed.

Let's work through a quick example using the Rails console:

```

>> l = Location.new(name: 'Barcade',
?> street: '388 Union Avenue',
?> city: 'Brooklyn',
?> state: State.find_by_abbreviation('NY'),
?> zip: '11211')
>> l.save
>> l
#<Location:0x007fa815acb8a8> {
  :id => 13,
  :name => "Barcade",
  :description => "blah",
  :created_at => Thu, 06 Nov 2014 15:48:15 UTC +00:00,
  :updated_at => Thu, 06 Nov 2014 15:48:15 UTC +00:00,
  :street => "388 Union Avenue",
  :city => "Brooklyn",
  :zip => "11211",
  :latitude => 40.7120412,
  :longitude => -73.95101339999999,
  :state_id => 33,
  :category_id => nil,
  :telephone => nil,
  :slug => "barcade",

```

```

      :published => true,
      :url => nil,
      :has_menu => false
    }

```

As you can see, once the record was saved the Geocoder was contacted, and the coordinates 40.7120412 and -73.9510133 were returned and inserted into the database.

Geocoding User Locations

After having geocoded a few arcades, we can begin geocoding site visitor's locations and determining whether the user is in close proximity to any of the location found in the database. Once Geocoder is installed a `location` method is made available to your Rails application's request object, meaning determining the location of any HTTP request is as simple as this:

```
location = request.location
```

The return location object contains a wealth of information about the user, including the user's country, region (state in the U.S.), city, zip code, and coordinates, among other attributes:

```

#<Geocoder::Result::Freegeoip:0x000000052041a8 @data = {
  "ip"=>"184.57.29.59",
  "country_code"=>"US",
  "country_name"=>"United States",
  "region_code"=>"OH",
  "region_name"=>"Ohio",
  "city"=>"Columbus",
  "zipcode"=>"43215",
  "latitude"=>39.9653,
  "longitude"=>-83.0235,
  "metro_code"=>"535",
  "area_code"=>"614"},
@cache_hit=nil>

```

Therefore to retrieve for instance the user's longitude you can access the returned object's data hash:

```

location = request.location
longitude = location.data['longitude']

```

Keep in mind the FreeGeoIP service limits you to 10,000 queries per hour. If you expect significant traffic, consider downloading and hosting a copy of the FreeGeoIP database and accessing the database directly. More information is available on the [FreeGeoIP¹⁹¹](http://freegeoip.net/) website.

¹⁹¹<http://freegeoip.net/>

Calculating Visitor Proximity to Locations

OK so now we have the coordinates for several arcades and know how to retrieve a visitor's coordinates, we can determine whether any locations are within a desired proximity of users. This is actually incredibly easy to do, because the Geocoder gem implements the [Haversine formula](http://en.wikipedia.org/wiki/Haversine_formula)¹⁹² for you. All you need to do is call the `near` method on a Geocoder-enabled model, passing in an array containing the user's coordinates:

```
user_location = request.location

locations = Location.near(
  [user_location.data['latitude'], user_location.data['longitude']]
)
```

The `near` method will by default return objects located within a 20 mile radius of the supplied coordinates. You can change the radius by passing in a value as the second parameter:

```
Location.near([latitude, longitude], 10)
```

Other options are available, such as using kilometer instead of miles. Consult the Geocoder documentation for more details.

Summary

There are plenty of powerful Active Record-related gems out there, and this short summary hardly scratches the surface. Over time I'll expand this chapter to introduce other gems such as Acts As List, Acts As Votable, and Awesome Nested Set. In the meantime be sure to e-mail me if you have any recommendations regarding gems I might include in this chapter!

¹⁹²http://en.wikipedia.org/wiki/Haversine_formula