Engine Yard University

Zero to Rails 3

Unit 1: Intro to Ruby

Unit Objectives

After completing this unit you should be able to:

- Explain the distinctive features of Ruby
- Read and write Ruby code
- Understand the features of the application we will be building in the labs

Unit Topics

- What is Ruby
- Ruby Basics

What is Ruby

Ruby is an interpreted, object-oriented language designed by Yukihiro Matsumoto, known to Rubyists as "Matz." Some describe it as a cross between Smalltalk and Perl. It is concise but readable. It is a pure object-oriented language with no primitives, i.e. "everything is an object." It features classical inheritance, mixins (a way to solve some of the problems that multiple inheritance solves), dynamic typing (aka "duck typing"), the ability to modify objects (including classes) at runtime, and elements of functional programming, notably blocks and closures. Ruby's dynamic nature makes metaprogramming easy, and Rails uses these features liberally.

Among the reasons Ruby is beloved by its partisans is its thriving ecosystem of developers whose communities encourage the sharing and discussion of code. Ruby is sometimes described as maximizing "developer happiness," and it is true that you will hear Rubyists speak of their language with an affection not often expressed towards other languages.

Ruby

Statements

Ruby statements are separated by line breaks. Semicolons are not typically used except to separate multiple statements on a single line.

Line breaks can also be used for readability after an operator, comma, or dot (prior to a method call).

Strings can also contain line breaks.

Data Types

Everything in Ruby is an object, there are no primitives as in some other languages (e.g. C++ and Java). Strings and numbers are objects just like arrays and hashes. Constants begin with a capital letter (you will note that classes are thus constants), and constants used to represent values are typically all uppercase, though this is not syntactically required. Note that some constants, like class names, are references; this means that while they will always refer to the same object, that object itself may be mutable. Symbols are prefixed with a colon. Arrays are delimited by brackets. Hashes are delimited by curly braces and keys are associated with values using the hashrocket (=>).

Strings

Strings are a series of characters surrounded by quotation marks. You can use either single quotes (`'`), or double-quotes (`"`) to wrap your string. Using double-quotes will interpret special escaped characters such as \n (a newline) or \t (a tab), as well as allowing for interpolation of variables. For example:

```
puts "This is a string\nwith a newline embedded."
```

The string is the data inside the double quotation marks, and it is passing it to the puts method, which will display the string on the screen. This example will output the following:

```
This is a string
with a newline embedded.
```

Strings can interpret and print the value of ruby code inline to make things look nice, similar to how you do it in languages like Perl, by using #{} inside a double-quoted string.

```
name = "Fred"
puts "Hello, #{name}!"
```

Symbols

Symbols are like immutable strings and thus far more memory efficient than strings. Each instance of `purple` is a new object, where as each instance of `:purple` is exactly the same object.

According to noted Rubyist and author of Rake, Jim Weirich, a good rule of thumb is that if the *contents* of the object are important, use a string, and if the *identity* of the object is important, use a symbol. Symbols are used liberally throughout Rails precisely based on this latter guideline.

Strings and symbols are not interchangeable, however, because it is so common to use strings and/or symbols as hash keys, Rails' `hash_with_indifferent_access` extends the Hash class to allow the interchangeable use of strings and symbols as keys.

Arrays

Arrays are ordered lists of objects, denoted by surrounding the list with square braces and separating each item in the list with a comma. The interesting thing about arrays in Ruby is that you can mix and match objects; your arrays do not need to contain a singular type.

```
my_array = [ 1, "two", 3, "four" ]
```

You can initialize a new array using square braces like above, or by using `Array.new` like this:

```
my_array = Array.new
```

Once you have an array, you can add new items to it using the "append" or "push" operator:

```
my_array << "New element"
# => [ 1, "two", 3, "four", "New element" ]
```

You can iterate over elements of the array using methods from the `Enumerable` module such as `each`.

Hashes

Hashes are lists of key/value pairs, stored in the order of their insertion. They are denoted by curly braces. The keys and values are separated using one of two different syntaxes. The first syntax uses the "hashrocket" operator to point from key to value:

```
my_hash = { :key => "value", :chunky => "bacon" }
```

The second method, new in Ruby 1.9, allows for a more compact hash syntax for keys that are symbols. Instead you can write the above like this:

```
my_hash = { key: "value", chunky: "bacon" }
```

To add a new element to a hash, you can use square braces after the name of the hash to identify the key, and assign the value:

```
my_hash = Hash.new
my_hash[:key] = "value"
```

Distinctive Operators

In addition to the hashrocket (=>), newcomers to Rails might also be unfamiliar with:

- the general comparison (or "spaceship") operator (<=>) which compares two values and returns `-1` if `a < b`, 0 if `a == b`, and 1 if `a > b`,
- and the "alligator" (<<) which is synonymous with append/push.

Note: with the exception of the hashrocket, the other operators are actually method names and can be redefined for other purposes.

Classes and Instances

Here is an example of how classes are defined and instantiated:

```
class Bar
  def initialize(name)
    @name = name
  end

  def greet
    puts "#{@name} says hi"
  end
end

foo = Bar.new("Ted")
foo.greet
```

The new method creates a new instance of a class, and calls the initialize instance method, if it exists. Other instance methods can be defined in the class and called by name using dot syntax.

Ruby's classes are open. This means they can be changed at runtime. This includes base classes like String:

```
class String
  def hello(name)
    "Hello, #{@name}"
  end
end

"pear".hello("Banana") # => "Hello, Banana"
```

Global variables, rarely used, are prefixed with `$`. Instance variables are prefixed with `@`; class variables with `@@` (note: classes can also have instance variables). Local variable have no prefix, and constant values, as noted previously, must begin with a capital letter and are all uppercase by convention (e.g. `DEFAULT = 0`).

Inheritance

Ruby classes can only inherit from a single class. Idiomatically, inheritance is primarily used when the subclass is a "kind of" the superclass. It is common to add behavior to a class using mixins, i.e. including modules.

Modules

Modules model qualities and abilities of things and cannot be instantiated directly, only included in classes. Class names are thus typically nouns (`Car, AmphibiousTank`), whereas module names are typically adjectives (`Driveable, Submersible`). Enumerable is an example of a powerful and frequently used module that adds traversal and searching methods such as `collect, find, reject,` and `sort` to collection classes like `Array` and `Hash`.

```
module Driveable
  def drive(speed=55)
    puts "Zoom, going #{speed} miles per hour"
  end
end

class Car < Vehicle
  include Driveable
end
```

Modules can also be used to namespace classes. For example:

```
module Foo
  class Bar
  end
end
```

Methods

Method calls are actually messages sent to a receiver, so `bar.foo` is the same as `bar.send(:foo)`, which could also be expressed as `some_method = :foo; bar.send(some_method)`. Parentheses can be omitted after method calls even when the method takes arguments, as long as there is no ambiguity.

Class/Instance Methods

Class methods can only be called directly on the class, while instance methods can only be called on instances. They are defined respectively as follows:

```
class Bar
  def self.method1
    puts "This is a class method"
  end

  def method2
    puts "This is an instance method"
  end

  class << self
    def method3
```

```
      puts "This is also a class method"
    end
  end
end
```

Public, Protected, and Private Instance Methods

Instance methods defined in Ruby, by default, are publicly accessible, but you can control the access of methods by using the `protected` or `private` keywords.

- **Private** methods are internal to the specific implementation of the class. As such, they can only be called by instances of the class.

- **Protected methods** are like private methods, but may be invoked by instances of the class or its subclasses.

Here is an example of a class with public, protected, and private methods:

```
class Foo
  def public_method
    "Anyone may call me."
  end

  protected

  def protected_method
    "I can be called by subclasses."
  end

  private

  def private_method
    "Only Foo may call me."
  end
end
```

Return Values

All methods return values (nil by default). The last expression evaluated in a method is the return value. As a result, the return keyword is often not used (also for performance reasons when using MRI. http://tomafro.net/2009/08/the-cost-of-explicit-returns-in-ruby

Arguments

Default arguments can be specified anywhere arguments are specified.

```
class Bar
  def foo(text="Hi!")
    puts text
  end
end
```

Duck Typing

Ruby objects are dynamically typed. Objects can be queried as to what they are with the `is_a?` method or which methods they respond to with the `respond_to?` method.

Blocks, Procs, and Lambdas

Blocks of code in Ruby are first-class citizens. You will see use of blocks all throughout Ruby and Ruby on Rails code, so it is important to understand what they are and how to use them. Using blocks of code in Ruby is one of the killer features that make the language so versatile and descriptive.

Blocks

Blocks are an anonymous grouping of code in Ruby that can be passed around and executed. You see blocks used frequently in Ruby, especially when dealing with `Enumerable` methods such as `collect` or `map`.

```
array = [ 1, 2, 3, 4 ]
array.collect! { |n| n+1 }
puts array.inspect
# [2, 3, 4, 5]
```

The code making up the contents of the block we pass to `collect!` -- it’s function code -- is between the curly braces. Any arguments passed to the block of code are denoted by a comma separated list of variables between the | characters. If you have a block that will have enough code to span multiple lines, you can do so with the above construct, but stylistically it is better to replace the curly braces with Ruby’s `do...end` keywords. For example, the above could look like this:

```
array.collect! do |n|
  n+1
end
```

Ruby methods accept blocks implicitly. When a block is passed to a method, it is executed by calling the `yield` statement, which temporarily transfers control to the passed code before continuing on with its execution. For example:

```
def fibonacci(maximum)
  i1, i2 = 1, 1
  while i1 <= maximum
    yield i1
    i1, i2 = i2, i1+i2
  end
end

fibonacci(100) { |f| print f, " " }
```

In this example, the method `fibonacci(maximum)` is called with an argument as well as a block of code similar to one we constructed previously. Since the block of code passed has an argument, we pass the argument in the `yield` statement.

Procs

According to the Ruby Standard Library documentation, Proc objects are "blocks of code that have been bound to a set of local variables. Once bound, the code may be called in different contexts and still access those variables." In other words, Procs are how Ruby defines anonymous function objects.

A basic Proc looks like this:

```
my_proc = Proc.new { |n| n+1 }
```

To invoke this function, use the call method on the Proc object, like this:

```
my_proc.call(5)
```

That executes the block of code and populates the variable `n` with the number 5.

Ruby 1.9 has a more compact syntax for defining Procs as well, using the "stabby proc" method:

```
my_proc = -> n { n+1 }
```

The above is equivalent to the first Proc we defined, with the argument list coming before the block of code instead of being defined inside it between two vertical bars. There is also multiple ways to call a Proc now:

```
my_proc.call(5)
my_proc[5]
my_proc.(5)
my_proc === 5
```

All of those methods are equivalent and call the Proc object.

Lambdas

Lambdas are very similar to a Proc. In both cases, the block of code passed to it becomes a callable object with access to the `call` method, but there are two differences between Procs and lambdas.

With lambdas, the number of arguments in the block of code are checked. Proc silently discards extra arguments. For example:

```
my_lambda = lambda   { |x,y,z| puts x*y*z }
my_proc   = Proc.new { |x,y,z| puts x*y*z }

# Prints 6 as expected
my_lambda.call(1,2,3)

# Fails with an ArgumentError
my_lambda.call(1,2,3,4)

# What does this return?
my_proc.call(1,2,3,4)
```

Calling `return` in lambda behaves differently than it does in a Proc. In a Proc, when using `return`, the method calling the Proc will stop execution and return. This is identical to the behavior of using a `return` with a Ruby block. With lambdas you get diminutive returns, meaning that it will return the value back to the method and it will continue on with its work. For example:

```
def invoke_proc
  Proc.new { return "Returning from a Proc." }.call
    "Greetings from invoke_proc"
  end

  def invoke_lambda
    lambda { return "Returning from a lambda." }.call
    "Greetings from invoke_lambda"
  end
end
```

The reason for this difference in behavior is traced back to the difference between procedures and methods. Procs behave like blocks when invoked, while lambdas are invoked like methods. The Proc object in Ruby is a snippet of code that is executed. When you run the `call` method on a Proc, you are running code directly in your calling method. So if it issues a `return`, it is in the scope of your calling method and as such it will return.

Lambdas, on the other hand, act just like anonymous methods. They check the number of arguments and return control back to the caller when finished.

Procs as Method Arguments

One final thing to discuss regarding Procs is their use as method arguments. According to the book Programming Ruby, "If the last argument to a method is preceded by an ampersand, Ruby assumes that it is a Proc object. It removes it from the parameter list, converts the Proc object into a block, and associates it with the method."

What this means is that you can pass a Proc by name as a final argument to a method by prefixing it with an ampersand and it will convert it to a block. For example:

```
length_selector = Proc.new { |word| word.length >= 5 }
[ "a", "short", "monsterous", "string" ].select(&length_selector)
```

is the same as:

```
[ "a", "short", "monsterous", "string" ].select { |word| word.length >= 5 }
```

Now, this trick gets very useful when you learn that not only will Ruby convert the argument to a block if it starts with an ampersand, but it will also call `to_proc` on that object first. How is this useful? It allows you to write code like this:

```
[ "a", "short", "monsterous", "string" ].map(&:capitalize)
# Returns => ["A", "Short", "Monsterous", "String"]
```

Woah. Wait? What happened? Ruby called `to_proc` on `:capitalize`, which is a Symbol. In Ruby 1.8.7+, `Symbol#to_proc` creates a block of code and calls the method name based on the symbol for the calling object. What you get is basically shorthand for this:

```
[ "a", "short", "monsterous", "string" ].map { |word| word.capitalize }
```

So it calls `.capitalize` on each element. Since each element here is a String, it has that method available to it and it capitalizes the string. Now you can impress your friends and coworkers with this bit of advanced Ruby-fu!

Ruby Version Manager

Rubyists tend to work with number of Ruby versions and implementations of the language. Instead of manually installing multiple versions, you can take advantage of the Ruby Version Manager, RVM, to install and quickly switch between multiple versions of Ruby on your system.

RVM also has the benefits giving you compartmentalized independent Ruby setups. This means that Ruby, gems and irb are all separate and self-contained from system and from each other. You can even have multiple named gemsets to keep gems from one project from interfering with another.

To install RVM, go to http://rvm.beginrescueend.com and follow the install instructions. For most systems, it is as easy as doing this, assuming you have bash and git installed:

```
$ bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

NOTE: if you are using the EYU learning environment, RVM is already installed and ready to go. You won't have to set it up on the provided instances.

Installing new rubies and switching between them

Let's say you have Ruby 1.8.7 on your system and you want to install Ruby 1.9.2. To install it with RVM, you would do the following:

```
$ rvm install 1.9.2
```

After a few minutes, it will have downloaded, configured, compiled, and installed Ruby 1.9.2 for you. Then you can use it by doing this:

```
$ rvm 1.9.2
$ ruby --version
ruby 1.9.2p180 (2011-02-18 revision 30909) [x86_64-darwin10.7.0]
$ rvm 1.8.7
$ ruby --version
ruby 1.8.7 (2011-02-18 patchlevel 334) [i686-darwin10.6.0]
```

To make 1.9.2 your default, you can do this:

```
$ rvm 1.9.2 --default
```

To see what rubies you have on your system:

```
$ rvm list

rvm rubies

   jruby-1.6.1 [ darwin-x86_64-java ]
   rbx-head [ x86_64 ]
   ree-1.8.7-2011.03 [ x86_64 ]
=> ruby-1.8.7-p334 [ x86_64 ]
   ruby-1.9.2-p0 [ x86_64 ]
   ruby-1.9.2-p180 [ x86_64 ]
```

You can get back to the system-installed ruby by typing "`rvm system`"

Gemsets

From the documentation for RVM:

"RVM has an extremely flexible gem management system called Named Gem Sets. RVM's gems(ets) make managing gems across multiple versions of ruby a non issue. RVM enables you to add a small text file to your application's repository instead of checking in tons of gems inflating your repository size needlessly. Additionally RVM's gemset management uses a common cache directory so as to only have one downloaded version of each gem on disk rather than several copies.

RVM helps ensure that all aspects of ruby are completely contained within user space, strongly encouraging non-root usage. Use of RVM rubies provide a higher level of system security and therefore reduce risk and overall system downtime. Additionally, since all processes run as the user, a compromised ruby process will not be able to compromise the entire system."

Creating a new gemset is easy:

```
$ rvm gemset create my_new_gemset
'my_new_gemset' gemset created (/Users/tim/.rvm/gems/ruby-1.8.7-p334@my_new_gemset).
$ rvm gemset use my_new_gemset
```

If you list the gems after executing "rvm gemset use" you will see a minimal set of gems installed on the system.

```
$ gem list

*** LOCAL GEMS ***

rake (0.8.7)
```

To view which ruby and gemset you are currently using, use the "current" command which shows both:

```
$ rvm current
ruby-1.9.2-p180@my_new_gemset
```

RVMRC

RVM utilizes three different bash scripts for configuration purposes.

- */etc/rvmrc* - Loaded before RVM initializes and before the user's ~/.rvmrc. This is used for all users on the system.
- *~/.rvmrc* - Used by a particular user to override settings in /etc/rvmrc.
- *Project .rvmrc* - This is what is used most often. You place a .rvmrc file in your project's root directory and it will tell RVM which ruby and gemset to use, as well as executing any other initialization code you wish.

NOTE: The system and user config files (/etc/rvmrc and ~/.rvmrc) should not contain any shell commands. Instead they should be used to set environment variables. Control rvm itself with the project .rvmrc

If you have a project named "rvm-tutorial" inside your Projects directory, and you wanted to take advantage of an .rvmrc to control RVM, you could do this:

```
$ cd ~/Projects/rvm-tutorial
$ echo "rvm 1.9.2@rvm-tutorial --create" > .rvmrc
$ cd ~/Projects/rvm-tutorial
$ rvm current
ruby-1.9.2-p180@rvm-tutorial
```

The first time you change into a directory with a new .rvmrc file in place, RVM will warn you and ask you to inspect it before RVM will trust it. After that, it will automatically execute the code in the file for you.

Unit Review

- Ruby
- Ruby features
- Syntax
- Statements
- Data Types
- Strings
- Symbols
- Strings v. Symbols
- Arrays
- Hashes
- Distinctive Operators
- Classes and Instances
- Variables
- Inheritance
- Modules
- Methods
- Class/Instance Methods
- Public, Protected, and Private Instance Methods
- Return Values
- Arguments
- Duck Typing
- Block, Procs, and Lambdas
- Ruby Version Manager

Lab 1

In this lab you will run irb (interactive ruby) and evaluate a few Ruby expressions.

Objectives

Get familiar with some Ruby basics.

Steps

The things you will type are in **bold**.

1. Run IRB:

   ```
   $ irb
   ```

2. You should see a Ruby prompt (yours may look a little different).

   ```
   >>
   ```

3. Let’s evaluate a simple expression:

   ```
   >> 1 + 2
   => 3
   ```

4. In Ruby, integers are objects. Let’s see what methods the integer 1 knows about:

   ```
   >> 1.methods
   ```

   ```
   => ["%", "odd?", "inspect", "prec_i", "<<", "tap", "div", "&", "clone", ">>", "public_methods", "object_id",
    "__send__", "instance_variable_defined?", "equal?", "freeze", "to_sym", "*", "ord", "+", "extend", "next", "send",
    "round", "methods", "prec_f", "-", "even?", "singleton_method_added", "divmod", "hash", "/", "integer?", "downto",
    "dup", "to_enum", "instance_variables", "|", "eql?", "size", "instance_eval", "truncate", "~", "id", "to_i",
    "singleton_methods", "modulo", "taint", "zero?", "times", "instance_variable_get", "frozen?", "enum_for", "display",
    "instance_of?", "^", "method", "to_a", "+@", "-@", "quo", "instance_exec", "type", "**", "upto", "to_f", "<",
    "step", "protected_methods", "<=>", "between?", "==", "remainder", ">", "===", "to_int", "nonzero?", "pred",
    "instance_variable_set", "coerce", "respond_to?", "kind_of?", "floor", "succ", ">=", "prec", "to_s", "<=", "fdiv",
    "class", "private_methods", "=~", "tainted?", "__id__", "abs", "untaint", "nil?", "chr", "id2name", "is_a?", "ceil",
    "[]"]
   ```

5. It can be hard to find what you’re looking for in that group, so let’s sort by name and also let’s remove the methods common to all objects.

   ```
   >> 1.methods.sort - Object.new.methods
   ```

   ```
   => ["%", "&", "*", "**", "+", "+@", "-", "-@", "/", "<", "<<", "<=", "<=>", ">", ">=", ">>", "[]", "^", "abs",
    "between?", "ceil", "chr", "coerce", "denominator", "div", "divmod", "downto", "even?", "fdiv", "floor", "gcd",
   "gcdlcm", "id2name", "integer?", "lcm", "modulo", "next", "nonzero?", "numerator", "odd?", "ord", "power!", "prec",
   "prec_f", "prec_i", "pred", "quo", "rdiv", "remainder", "round", "rpower", "singleton_method_added", "size", "step",
   "succ", "times", "to_f", "to_i", "to_int", "to_r", "to_sym", "truncate", "upto", "zero?", "|", "~"]
   ```

6. Let’s write ’hello world’.

   ```
   >> def hello
   ```

```
>> puts "Hello world!"

>> end

=> nil

>> hello

Hello world!

=> nil
```

7. The reason just typing `hello` works is because methods without an explicit object are run against self. In this case, we're in an instance of Object.

```
>> self.class

=> Object
```

8. When we defined our method, it was defined for the Object class. This means that all of Object's ancestors now have the class. That can seem a little strange at first, but it illustrates how easy it is to change even Ruby's base classes. Typically, though, when writing an application, we won't be in the this context (e.g. when we're creating a new class, the context will be our class).

```
>> 1.hello

Hello world!

=> nil

>> "kitty cat".hello

Hello world!

=> nil

>> String.hello

Hello world!
```

9. Experiment with using RVM to switch between Ruby versions. Type `rvm list` to see a list of installed Ruby versions. An arrow (=>) will be pointing to the current Ruby for your console session, which should be Ruby 1.9.x. You can also enter `which ruby` to see which version of Ruby is in place. Use `rvm ruby-1.8.7` to switch to Ruby 1.8.x, and confirm that the switch occurred using `rvm list` or `which ruby`. In Ruby 1.8.x accessing a character at a particular index of a String will return the ASCII code for the character. Try using IRB, and typing `"Ruby"[2]`.

   Now exit the IRB console and switch back to Ruby 1.9.x using `rvm ruby 1.9.2`. Start IRB again. When you enter `"Ruby"[2]` you should see the character 'b'.

10. Experiment with Enumerable methods that take a Block. Define `my_array` with the value of `[0,10,20,30,40,50,60,70,80,90,100]`. Use the `delete_if` method to generate an Array containing numbers >= 50. The documentation for Array and its methods is here: http://www.ruby-doc.org/core/classes/Array.html

11. Developers often specify options in Hash form. Define a hash of default options for a car:

    ```
    default_options = { :color => 'Red', :seats => 4,
      :air_conditioning => true, :power_steering => true,
      :transmission_type => "auto"}
    ```

    Now, specify a hash of our specific options for a given car:

    ```
    edsel_options = { :color => 'Blue', :seats => 6 }
    ```

    Finally, merge the specific car's options with the default options:

    ```
    default_options.merge(edsel_options)
    ```

    Any options defined in the specific car's options override the corresponding default options hash. Take a look at the resulting hash returned by the `merge` method. You now have a full set of car options where the `default_options` elements fill in any values not supplied by the `edsel_options` hash.

    This technique is used by a number of Rails components, such as date selection fields, to keep the user from having to specify a full set of properties. An example of a method that takes a set of options, specified via hash, is `DateHelper.date_select`. Take a look at the DateHelper documentation for this class: http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html#method-i-date_select.

12. Create a method that prints out a string and takes two arguments: a string and a block.

    The method's output should contain all the characters that appear in the string, but the method output and the supplied string should differ in format. For example, the string passed in might be "Abcd", but the output might be reversed (ie "dcbA") or the output might be all lowercase ("abcd"). The output can contain more characters than the supplied string, but not less. If "Abcd" is supplied, the output could be "a*b*c*d".

    The block should be used to format the output, but should not modify its string argument. The documentation for the String class describes a number of methods that can be used to format a String: http://ruby-doc.org/core/classes/String.html

Unit 2: Discovering Rails

Unit Objectives

After completing this unit you should be able to:

- Explain the features and limitations of the Rails framework
- Understand the concepts underlying web applications
- Generate a bare bones Rails application
- Know about Ruby Gems and how to install them using Bundler.
- Start the Rails server and view server logs

Unit Topics

- What is Rails
- Rails Features and Limitations
- Web Applications 101
- Generating a Rails application
- Running a Rails application

What is Rails

Rails is a web application framework built on Ruby. Rails was developed at 37signals and extracted from their products and released as an open source framework. The combination of Rails and Ruby provides for rapid development and iteration with relatively little overhead.

Other Frameworks

There are dozens of web frameworks, each with its partisans and detractors, and Rails is no exception. There are several other frameworks built on Ruby, such as Sinatra and Merb. Merb was merged into Rails, producing the Rails 3 release in 2010. Django is a popular framework written in Python; PHP offers many frameworks including the venerable Zend, Symfony, and the Rails-inspired CakePHP; even languages with narrower bases like ColdFusion and Scala have their own frameworks.

Rails Features and Limitations

Rails is both popular (some would even say "trendy") and also the subject of some criticism. Among the most common critiques of Rails are that it is slow, and that its codebase has become too large; that it forces developers to do things its way; and that it was not built with a full understanding of relational databases, discouraging developers, for example, from writing their own SQL.

Speed and Scaling

It is true that there are faster languages than Ruby, and if speed is the most important feature of your application, you might be better served by other frameworks, or no framework at all. That said, Rails offers a good balance between ease and speed of development and performance. Twitter, famously, had serious performance problems in 2008 which were partly blamed on Rails (justifiably or not), and the reality is that if you need to process as much data as quickly and at the volume that Twitter does, the speed of Rails may well become an issue.

There are many Ruby interpreters, some of which perform considerably faster than others. JRuby and Ruby 1.9 are both considerably faster, for example, than stock Ruby 1.8 (often referred to as MRI â Matz’s Ruby Implementation).

The Rails Way

The 37signals team is opinionated and not afraid to show it. They describe Rails as "opinionated software," which can be a good thing as well as a limitation. By assuming that there is one good way to do most things (while still allowing you to do it differently if you choose), Rails eliminates much of the setup and configuration that is required in some other frameworks, saving you, the developer, precious time and lines of code.

Rails and SQL

One of the benefits of a framework is specifically the abstraction of the database. In that sense not having to write SQL is a feature of Rails, not a limitation. There is no technical impediment to writing SQL in Rails, it’s just that in most cases you will not want or need to.

Web Applications 101

MVC

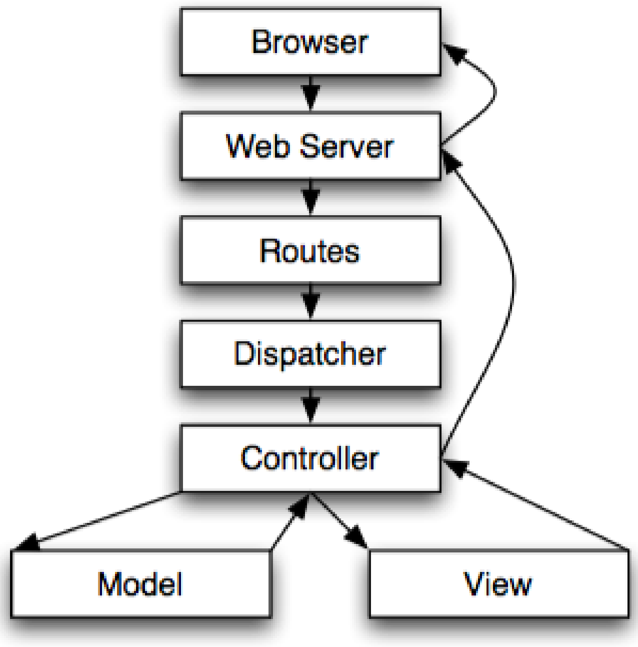
Many modern database-backed web application frameworks are based on the Model/View/Controller architecture pattern, which is composed of several design patterns and first articulated in the Gang of Four's classic Design Patterns. Bear in mind that MVC as used in embedded systems or iOS, for example, can be different than its implementation in Rails.

MVC and Rails

In Rails, the expression "fat models" refers to the fact that models are intended to do most of the heavy lifting. The bulk of the application logic should be located in the models. Controllers, which handle the interaction between models and views, should typically do very little. As a rule of thumb, if your controller methods begin to exceed 10 or so lines, it might be a sign that refactoring may be appropriate. See Skinny Controller, Fat Model" by Jamis Buck for more information.

Views are intended to handle presentation, and should contain as little logic as possible. Presentation logic, when needed, can be extracted into view helpers for even cleaner, simpler views. Models and controllers are both classes (subclasses of ActiveRecord and ActiveController respectively) while views are HTML templates.

Request Lifecyle



Unit Review

- Rails
- Rails features
- Rails limitations
- Speed and scaling
- Convention over configuration
- Rails and SQL
- MVC
- Requests and MVC

Lab 2: Creating and running a rails application

In this lab you will create a simple rails application and run it.

Objectives

After completing this lab, you should be able to:

- Create a rails application
- Run the application using rails' built-in server
- Specify and install gems required for the application
- Create and view documentation for the application
- Create and view documentation and guides for rails

Steps

Create a new application

1. Download the application template file from **`https://raw.github.com/engineyard/traveljournal/master/app-template.rb`**.

2. Generate the application skeleton using a pre-defined application template. This template sets up various components for us such as RSpec for testing, HAML for views, and jQuery for Javascript. To generate the application using the template, run this command:

```
$ rails new traveljournal -T -m app-template.rb
```

The above call to "rails new" uses several command line arguments. Let's look at some of these briefly:

- **-T** Skip generating files for Test::Unit. We're using RSpec instead.
- **-m <template>** Use our template script to aid in generation of the project.

You can see more options by running this command: `rails new -h`

Begin configuring the application

1. Run bundler (only if running on Rails 3.0. On Rails 3.1 the rails setup command runs Bundler automatically. You should at least try it to see how it works).

```
$ cd traveljournal
```

`$ bundle install` or, with the most recent versions, **`bundle`**, as the default action is install.

2. Bootstrap RSpec, the testing framework we will use for this course.

```
$ rails g rspec:install
```

3. Review generated files.
4. Run WEBrick server.

   ```
   $ rails s
   ```

5. View the Application and Logs
6. Check app in browser by loading the url: **http://localhost:3000**
7. Hit refresh in your browser, and watch your terminal as you browse the page.

Create and View Documentation

1. Create rails docs with rake.

   ```
   $ rake doc:rails
   ```

2. Create rails guides with rake.

   ```
   $ rake doc:guides
   ```

Generating guides requires RedCloth

If you didn't use the Engine Yard application template, you will see this error pop up when executing ***rake doc:guides****:*

```
rake aborted!
no such file to load -- redcloth
```

1. Please add

   ```
   gem 'RedCloth', '~> 4.2.7'
   ```

2. to the Gemfile, then run

   ```
   bundle install
   ```

3. and try again.
4. Oops! We get an error. Follow the instructions in the console and edit the Gemfile
5. Run bundle install again.

   ```
   $ bundle install

   [...] # output from command

   $ rake doc:guides
   ```

6. Create app docs with rake.

```
$ rake doc:app
```

Now you can view documentation and guides for Rails in `~/Projects/traveljournal/docs`. For online references check out the following resources:

- Ruby Docs Online - http://www.ruby-doc.org/
- Rails Docs Online - http://rubyonrails.org/documentation
- Rails Guides Online - http://guides.rubyonrails.org/
- Railscasts - http://railscasts.com/
- ASCIICasts - http://asciicasts.com/

7. We talked about the concept of "Open Classes" in Ruby. The Rails Core team added several methods to Ruby Core classes to make life easier for Rails developers and to make the Rails framework easier to maintain. These modifications to Ruby are located in the ActiveSupport module. By default all of the ActiveSupport features are included in a new Rails application when it is generated, but it's possible to cherry pick particular ActiveSupport submodules. Let's try out some of the ActiveSupport methods in the Rails console.

   Start a Rails console session with `rails c` - this is short for `rails console`.

   Try out ActiveSupport's relative date calculations, which are implemented via enhancements to Ruby's Fixnum class. Type `2.days.ago`. Try `4.years.from_now`.

   Try out some of the following Rails-style enhancements to the Date class (you can access the current date in your time zone via `Date.current`:

   ```
   Date.current.prev_year
   Date.current.prev_month
   Date.current.beginning_of_week
   Date.current.next_week
   Date.current.end_of_week
   Date.current.end_of_quarter
   ```

   The Rails ActiveSupport Guide is a great reference: http://guides.rubyonrails.org/active*support*core_extensions.html

8. ActiveSupport's `blank?` method, implemented as an instance method for Strings, is useful for evaluating parameters submitted via web form. It returns true is the String is either `nil` or blank (ie ""). Try the following: str=nil str.blank? str="" str.blank? str="hello" str.blank?

9. How easy is modify Ruby when working in the Rails environment? In this exercise we'll add a method to the String class and access it from the Rails console.

   Exit your console session.

   Create a new file in the `config/initializers` directory called `core_extensions.rb`

   Open the String class by adding code within a local version of the class definition. Use the following as a guide for creating your own enhancement to the String class and paste it into the `core_extensions.rb` file.

```
class String
  def my_custom_method()
    put "Custom output!"
  end
end
```

Restart the console using the `rails c` command.

Try out the new String enhancement using code like:

```
"ABC".my_custom_method
```

10. As a reality check, open an `irb` session and try some of the methods mentioned in the ActiveSupport exercises outside of the Rails environment. You should not be able to successfully call `days_ago` on a number or `blank?` on a String.

Unit 3: A bit about Git

Unit Objectives

After completing this unit you should be able to:

- Explain the difference between distributed and centralized version control
- Explain the difference between the working directory, the index and the repository
- Add and commit new code locally
- Check status
- Reset to a known state
- Switch branches

Unit Topics

- The What and Wherefore of Git
- Git Basics

The What and Wherefore of Git

In this class, we will be using the Git version control system to make sure we are all in sync at the start of each unit. Why Git? Mostly because almost every Ruby project of note uses it for version control, but also because it is a great version control system. For one thing, Git allows each student of this class to have a copy of the source code on their computer (including various branches). These local repositories can be accessed without a network connection, and can be changed independently of one another.

Distributed Version Control

If you are not used to distributed version control, the concept above might sound a little strange. Distributed systems differ from older centralized systems like Subversion or CVS in that each team member can have a copy of the entire repository on their computer (along with their individual changes). They also facilitate sharing updates with each other. Git supports several protocols, including HTTP, email, and it's own protocol.

Although Git is distributed, most Ruby projects use a centralized distributed model. That means that one repository is designated as canonical (e.g. GitHub, a popular hosting service that we will be using in this class) and the others (e.g. developers) push changes to and pull changes from the central repo. Note: Although this is a common pattern for Rails projects, other projects are more distributed. Git was originally invented by Linus Torvalds to manage the Linux kernel, and the usage of Git by kernel developers is considerably different.

Git Basics

Git is a very complex tool and could take up an entire class on its own. Here we will just cover the basic commands we will need in this class. This is a walkthrough of some basic things we may be doing with git. The instructions here assume that you have followed the instructions you were sent to setup your computer with the environment that we will be using for the class.

Notes for Windows Users

Ruby, Git, and other software we will be using in this class come out of a Unix-influenced culture. While a lot of work has gone into making things work well on Windows, you may encounter a few unfamiliar things on your machines. The first is that instead of running the regular command prompt (cmd.exe), we will be using git-bash. Bash is a popular shell on Unix-like operating systems. To use this shell, you will need to run git-bash.bat (by default it is installed in c:\RailsInstaller\Git). Since we will be using this frequently, we suggest creating a shortcut:

1. In Explorer, navigate to c:\RailsInstaller\Git.

2. Find git-bash.bat (if you have extensions hidden, you will see it as simply git-bash). Double-click it. If it is the right file, it will open up a console with a $ prompt.

3. Drag the file icon to the Start menu.

Terminology

Here are some basic terms that we will be using. These are simplified for the purposes of this class.

- **Commit** -- A commit is a single point in time for the code. Each commit can be thought of as a patch (aka, a difference between the current state and the previous state) plus a link to the parent commit if it has one. A commit can contain changes to more than one file, which is identified by a hash (some other systems use ordinal numbers).
- **Repository** -- This is the entirety of a project. This includes all of the commits, and hence the entire history of the project (possibly including many branches).
- **Working directory** -- What we currently see on the file system, or the files that we are working with before we commit changes to the repository.
- **Index** -- Also called the staging area, the index is a place we can specify what is about to be committed before we commit.
- **Branch** -- Branches are essentially named commits. When new commits are made to a branch, the branch then points to the latest commit.
- **Tags** -- Like branches, tags are named commits. However, they refer to a specific commit and do not move forward as branches do.

Quick Walkthrough

1. Let's start by setting the user name we will be using for git. Use your name and e-mail address in place of the examples listed here.

   ```
   $ git config --global user.name "Happy Coder"

   $ git config --global user.email "happy_coder@example.com"
   ```

2. Next, let's create a new Git repository. Create a new directory and initialize it (for our examples, our base directory will be Projects inside the home directory, but you can do this in any directory you like):

   ```
   $ cd ~/Projects

   $ mkdir learning_git

   $ cd learning_git

   $ git init
   ```

3. We now have an empty repository. Let's add a file. First, create a README.TXT file in this directory using your preferred text editor or type these commands:

   ```
   $ cat > README.TXT
   ```

```
Hello world!
```

```
ctrl-d
```

4. Now let's stage it. Staging means putting the changes we want to commit in the Index. This is a temporary area where we can specify what we want to commit before we actually do the commit. Note: We can add the file by name, but we can also use directories. Since we only have one file, we can add the current directory (i.e. everything).

```
git add .
```

5. So far, no changes have been made to our repository -- the README.TXT file is staged but not yet committed. We can see the current status like so:

```
$ git status

    # On branch master
    #
    # Initial commit
    #
    # Changes to be committed:
    #   (use "git rm --cached ..." to unstage)
    #
    #       new file:   README.TXT
    #
```

6. Now let's commit our change and look at status again.

```
$ git commit -m "Initial commit."

    Initial commit.
    1 files changed, 1 insertions(+), 0 deletions(-)
    create mode 100644 README.TXT

$ git status

    # On branch master
    nothing to commit (working directory clean)
```

Note: If you omit the -m "Initial commit." portion, git will open up your default editor. On many systems, this is Vim or vi. While Vim is a great editor, it is not the easiest editor to learn for those new to it. Fortunately, you can set what editor to use by setting the EDITOR environment variable.

For example, if you are using Notepad++ on Windows, you can do this:

```
$ set EDITOR="c:\Program Files\Notepad++\notepad++.exe"

$ git commit
```

After hitting enter, Notepad++ will open and you can enter your comment. Once you close the editor (not just the tab), git will finish the commit. Alternatively, you can use a Git GUI.

7. Now let's add another file and stage it.

   ```
   $ cat > hello_world.rb

   puts "Hello world!"

   ctrl-d

   $ git add hello_world.rb

   $ git status
   ```

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD ..." to unstage)
   #
   #       new file:   hello_world.rb
   ```

8. Before committing the file, though, let's make some changes:

   ```
   $ cat >> hello_world.rb

   puts "How are you doing?"

   ctrl-d

   $ git status
   ```

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD ..." to unstage)
   #
   #       new file:   hello_world.rb
   #
   # Changed but not updated:
   #   (use "git add ..." to update what will be committed)
   #   (use "git checkout -- ..." to discard changes in working directory)
   #
   #       modified:   hello_world.rb
   ```

   Note that "hello_world.rb" appears in both the "changes to be committed" and the "changed but not updated" sections. If you commit at this point, only the changes that are staged will be committed. Let's try that:

   ```
   $ git commit -m "This should only commit the first change."

   $ git status
   ```

   ```
   # On branch master
   # Changed but not updated:
   #   (use "git add ..." to update what will be committed)
   #   (use "git checkout -- ..." to discard changes in working directory)
   #
   #       modified:   hello_world.rb
   #
   ```

9. Let’s add this remaining change into a new branch. To checkout new branches, we use the ’git checkout’ command. When we add â-b’, this creates a new branch.

```
$ git checkout -b another_branch

M hello_world.rb

Switched to a new branch ’another_branch’

$ git add .

$ git commit -m "Committing this change to a new branch."

[another_branch bd245d4] Committing this change to a new branch.
1 files changed, 1 insertions(+), 0 deletions(-)

$ git status

# On branch another_branch
nothing to commit (working directory clean)
```

10. To list our branches, we can do:

```
$ git branch

* another_branch
    master
```

11. And to see the log, simply do:

```
$ git log

commit bd245d4ef51752169344ce5c54d57492d4eb673b
Author: Happy Coder happy_coder@example.com
Date:   Wed Feb 16 03:06:41 2011 -0800


Committing this change to a new branch.


commit becb4e32cc1e379a763b6c711e8f7a06d1db15c9
Author: Happy Coder happy_coder@example.com
Date:   Wed Feb 16 02:57:01 2011 -0800


This should only commit the first change.


commit 9f10d3697462de038e4069eaf3531e4a05989d70
Author: Happy Coder happy_coder@example.com
Date:   Wed Feb 16 02:29:53 2011 -0800


First commit.
```

12. Now let’s switch back to our first branch and see what the log looks like. It should have all the same same commits except for the most recent one above.

```
$ git checkout master

Switched to branch 'master'

$ git log

commit becb4e32cc1e379a763b6c711e8f7a06d1db15c9
Author: Happy Coder happy_coder@example.com
Date:   Wed Feb 16 02:57:01 2011 -0800


This should only commit the first change.


commit 9f10d3697462de038e4069eaf3531e4a05989d70
Author: Happy Coder happy_coder@example.com
Date:   Wed Feb 16 02:29:53 2011 -0800


First commit.
```

13. Let's say that we make a mistake and need to get back to a known state. This is easy with git. For example:

```
$ rm README.TXT

$ **git status

# On branch master
    # Changed but not updated:
    #   (use "git add/rm ..." to update what will be committed)
    #   (use "git checkout -- ..." to discard changes in working directory)
    #
    #       deleted:    README.TXT
    #
    no changes added to commit (use "git add" and/or "git commit -a")

$ git checkout README.TXT

$ git status

# On branch master
    nothing to commit (working directory clean)
```

Git Clients

Mac

- GitX
- TextMate bundle
- Tower

Windows

- Git for Windows (msysgit)
- TortoiseGit

Unit Review

- Git
- Repository, Working Directory and Index
- Checking out code
- Switching branches
- Staging changes
- Committing changes
- Resetting changes

Unit 4: Models

Unit Objectives

After completing this unit you should be able to:

- Create and maintain a database using Rails
- Write Models that describe your data objects using validations and associations
- Write Active Record queries using finder methods and scopes
- Use the DB console and Rails console

Unit Topics

- Managing Databases
- Models
- Seeds
- Using the Rails Console and DB Console
- Validating Model Data
- Mapping Object Relationships with Associations
- Queries

Managing Databases

ORM & Active Record

Database-backed web application frameworks benefit from encapsulation of database access. Without a framework, you have to handle the overhead of opening, querying and closing a connection to the database every time you access it. Object-relational mapping (ORM) handles database access overhead, as well as abstracting the database for your application, allowing you to change database vendors with minimal (but non-zero) effort, assuming you have not written any custom SQL that uses vendor-specific features.

Rails' default ORM, ActiveRecord, is a straightforwardly named implementation of the Active Record pattern described in *Martin Fowler's Patterns of Enterprise Application Architecture (PoEAA)*. It can easily be replaced with other ORMs, such as DataMapper (based on the Data Mapper pattern, also described in PoEAA), Sequel, or others. You can also choose to use adapters for non-relational databases, such as MongoDB or CouchDB.

ActiveRecord maps database tables to objects (using clever pluralization rules to handle most common cases, e.g. table `people` to class `Person`), rows to instances of those objects, and columns to methods of those instances. The ORM also adds a number of helpful methods for querying, validating data, and handling relationships. You will also add methods to your models that encapsulate much of your application logic.

Many of the features provided in ActiveRecord are actually mixed in by modules belonging to the ActiveModel namespace, introduced in Rails 3. These same modules can also be mixed into a standard Ruby class, thus providing useful behavior such as validation and serialization.

Database Configuration

Rails is set up with three environments by default: development, test, and production. Each environment uses a separate database, and each environment has a separate configuration file, which we'll look at in more detail in Unit 7.

Specific information about how to connect to your databases is stored in `config/database.yml`, which is created when you generate your rails application. Typically you will only need to edit this file to specify the access credentials rails will use to connect to the database, or if you change database vendors. Rails assumes a sqlite database by default, but when generating your application, you can specify the database type with the `-d` flag. The options include `mysql, oracle, postgresql, sqlite3, frontbase,` and `ibm_db`. Depending on your system, you may also need to install an architecture-specific database adapter Ruby gem. In this course we will use sqlite, which require no additional configuration at this time.

Rails includes a set of rake tasks to help manage creating and altering your databases. A good way to explore rake is to use the `-T flag`, which allows you to see all rake tasks matching a particular string. `rake -T db` will provide you with this list (including descriptive comments not shown here):

```
rake db:create
rake db:drop
rake db:fixtures:load
rake db:migrate
rake db:migrate:status
rake db:rollback
rake db:schema:dump
rake db:schema:load
rake db:seed
rake db:setup
rake db:structure:dump
rake db:version
```

When running rake tasks, particularly database-related tasks, prefix the rake command with the target environment.

To create your development database:

```
RAILS_ENV=development rake db:create
```

As a shortcut, to create databases for all your environments, there is a separate rake task:

```
rake db:create:all
```

Migration Basics

Versioning your database Data Definition Language (DDL)is a good idea and Rails migrations are a built-in way of doing exactly that. Migrations are the recommended way to create and alter tables as well as migrate data if necessary to accommodate schema changes. Migrations allow you to rollback or forward to any version of your schema. Previously, migrations were numbered sequentially, occasionally leading to conflicts between multiple developers working on the same project. Migrations are now timestamped, which minimizes the possibility of such collisions.

Note: Migrations also have access to your models. While using model methods to help with a complex migration can be tempting, it's a risky practice (since, if you're running a migration in the far future, your Ruby code might no longer be compatible with the migration).

Migrations are subclasses of ActiveRecord::Migration and contain an `up` method which defines the additions or changes, and a `down` method which defines what to do when rolling the migration back. In the event that your migration drops tables or columns, a rollback will, obviously, not be able to recover the lost data.

A migration to create a simple hotels table with two columns might look like this:

```
class CreateHotels < ActiveRecord::Migration
  def up
    create_table :hotels do |t|
      t.string :name
      t.text :description
      t.timestamps
    end
  end

  def down
    drop_table :hotels
  end
end
```

Migration Tips

When creating a table, for example, within the `up` method you will use a `create_table` block which will contain a list of columns and their data types. The down method will simply drop the table.

It is such a common paradigm of database migrations that the `up/down` pair can be simplified into a single `change` method to handle both up and down cases as long as the changes can be easily determined from the method. Your migration ends up looking like this:

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

If your migration involves a change that cannot easily be reversed, such as a `remove_column`, then you can fall back on using the `up` and `down` methods instead of `change`.

Rails' helper method `timestamps` creates two "magic" columns in the database, `created_at` and `updated_at`, which Rails will automatically update when you create and update rows.

You can also dump your entire database schema into a single file, `db/schema.rb`, using `rake db:schema:dump` which you should create and maintain in your version control system. When setting up a new development environment for an existing project, it is much faster to create your tables using `rake db:schema:load` rather than running all the migrations.

DB Console

The Rails DB console is a convenient way to access the command line for your database. Invoke the DB console using `rails db`. Once inside the console the syntax will be specific to the database you are using. In the case of sqlite, you can type `.help` to see a list of commands.

Walkthrough 4-1: Creating a Migration

In this walkthrough you will write a migration to create the hotels table with two columns. Steps

1. Create your development database.

   ```
   $ rake db:create
   ```

2. Use Rails' migration generator to create a migration. In this case, we are using the generator primarily to take advantage of the automatic timestamping. Note that the migration is its own class, different from the model class referred to by the table.

   ```
   $ rails g migration CreateHotels
   ```

3. Open the file that was generated in `db/migrate/`, which will have a filename similar to `20100112221858_create_hotels.rb`. The contents of that file should resemble this:

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


  def down
  end
end
```

Rails generates both an `up` and `down` stub when you perform a `rails g migration` command.

4. Remove the up and down method and replace with the change method, which is a Rails 3.1 feature that can shorten your code by not requiring you to define `drop_table`. Rails can intuit how to do the reverse operation (down) for most cases.

```
def change
  create_table :hotels do |t|
    t.string :name
    t.text   :description
    t.timestamps
  end
end
```

5. Save the file and run the migration You should see an indication that your table was created.

```
$ rake db:migrate

==  CreateHotels: migrating =========================
-- create_table(:hotels)
-> 0.0011s
==  CreateHotels: migrated (0.0012s) ================
```

6. Using the DB console, you should be able to verify that the table exists and has the desired columns. Open the db console. If you don't specify an environment, you will be accessing the development database (which in this case is what we want).

```
$ rails db

SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

7. List the tables

```
sqlite> .tables

hotels
schema_migrations
```

8. Dump the schema for the hotels table.

```
sqlite> .schema hotels


CREATE TABLE "hotels" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  "name" varchar(255), "description" text,
  "created_at" datetime, "updated_at" datetime);
```

The SQL should accurately reflect your migration. Note that Rails automatically creates an auto-incrementing primary key column id in addition to the timestamp columns.

Quit the DB console.

```
sqlite> .exit
```

9. Sometimes it's handy to be able to easily undo database changes made via migration. Undo the most recent migration using the command:

   ```
   rake db:rollback
   ```

   Rerun the migration and verify that the hotels table is back:

   ```
   rake db:migrate
   ```

   Note: If you need to rollback an earlier migration, you can specify the numeric timestamp prefix of the migration, using the `VERSION` option and a command such as:

   ```
   rake db:migrate VERSION=20110728435031
   ```

10. Rollback the migration that created the hotels table. Ensure that the table no longer appears in the table list maintained by sqlite. Rerun the migration and verify that the hotels table is back.

Models

Models are typically subclasses of ActiveRecord. Column mapping to methods is handled automatically. Models will typically contain:

- associations - information about relationships between objects
- validations - data integrity specifications
- additional custom methods describing your application logic

Some of the most common built-in methods used in working with models include:

- new - creates an instance without saving it to the database
- valid? - predicate that verifies the validity of an instance
- save - saves an instance created with new
- create - instantiates and saves an instance to the database
- find - queries the database

Walkthrough 4-2: Creating a Model

In this walkthrough you will create a simple model and practice creating and saving objects using the Rails console.

Steps

1. Create the class `app/models/hotel.rb`

```
class Hotel < ActiveRecord::Base
end
```

2. That's it!

Rails Console

The Rails console is similar to IRB in that it gives you an interactive ruby console, except that it also loads your entire application environment so that you have access to the additional features of Rails as well as your application code.

You can use the console, for example, to interact directly with the Hotel model we just created.

```
$ rails c
```

```
peapod = Hotel.new(:name => "Peapod Inn",
   :description => "A green and cozy place.")
#< Hotel id: nil, name: "Peapod Inn", description: "A green and cozy place.",
   created_at: nil, updated_at: nil>
> peapod.valid?
=> true
> peapod.save
=> true
```

Seeds

You can seed your database with data that you will need for your application. Seeds should not be used for test data. Instead, you can use plain-old Ruby code, fixtures, test doubles, factories. The `seeds.rb` file contains Ruby code that you can run separately with `rake db:seed`, or as part of a complete database bootstrap with `rake db:setup`.

Walkthrough 4-3: Creating Seeds

In this walkthrough you will write a migration to seed the database with cities. In a complete application your list of cities would be more comprehensive.

Steps

1. Use the Rails generators to create both a migration and a model for City with a single attribute, name.

   ```
   $ rails g model City name:string
   ```

2. Run the migration.

   ```
   $ rake db:migrate
   ```

3. Create `db/seeds.rb`

```
City.create([
    { :name => "Chattanooga" },
    { :name => "Chicago" },
    { :name => "Cincinnati" }
])
```

4. Load the seeds into the database:

```
$ rake db:seed
```

5. Verify that the seeds were loaded.

```
$ rails c
```

```
> City.all
```

Validations

Validations allow you to handle data integrity at the application level, as opposed to client-side (using Javascript) or at the database level (using constraints). Validations are normally triggered when the object is saved, but can also be specified to be triggered at any stage in the object life cycle.

Rails provides a host of validation helpers for common validations, including:

```
validates_acceptance_of
validates_associated
validates_confirmation_of
validates_exclusion_of
validates_format_of
validates_inclusion_of
validates_length_of
validates_numericality_of
validates_presence_of
validates_uniqueness_of
validates_with
validates_each
```

An example of a simple validation to ensure that a hotel has a name and that the name is at least two characters long:

```
class Hotel < ActiveRecord::Base
  validates :name, :presence => true
  validates_length_of :name, :minimum => 2
end
```

Notice that the first validation uses `validates :name` instead of the full `validates_presence_of` validator as shown on the list above. Rails 3 provides this as a shortcut to call any defined validator. This allows you to add multiple validators to a single line of code. This is perfectly acceptable - however, beware of long lines of code causing your model to become awkward to read. To add both the presence and length check in a single line, you can do something like this:

```
class Hotel < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 2 }
end
```

Validation Tips

Rails validations are carried out using the valid? method which can also be called directly. Assuming the validations specified above, the following should return false.

```
h = Hotel.create
h.valid?
```

Errors are accessible in the errors instance method after validations are triggered

```
h.errors
```

Adding a valid name should result in the valid? returning true.

```
h.name = "Joe's Inn"
h.valid?
```

Validations can be skipped:

```
h = Hotel.new
h.save(:validate => false)
```

Validation Gotchas

It can be hard at first to remember which model methods trigger validations and which do not.

Validations ARE triggered by:

```
    create
    create!
    save
    save!
    update
    update_attributes
    update_attributes!
```

Validations are NOT triggered by:

```
    decrement!
    decrement_counter
    increment
    increment_counter
    new
    toggle
    update_all
    update_attribute
    update_counters
```

The biggest surprise is that though `update_attributes` does trigger validations, `update_attribute` does not.

Walkthrough 4-4: Creating Validations

In this walkthrough you will add two validations to the Hotel model.

Steps

1. Add a validation to `app/models/hotel.rb`:

   ```
   class Hotel < ActiveRecord::Base
     validates :name, :presence => true
   end
   ```

2. Test the validation in the Rails console:

   ```
   $ rails c

   > hotel = Hotel.new
   > hotel.valid?
   => false
   > hotel.name = "The Peabody Inn"
   > hotel.valid?
   => true
   ```

3. Add a second validation.

   ```
   class Hotel < ActiveRecord::Base
     validates :name, :presence => true
     validates_length_of :name, :minimum => 2
   end
   ```

4. Test the validation in the Rails console:

   ```
   $ rails c

   > hotel = Hotel.new
   > hotel.name = "A"
   > hotel.valid?
   => false
   ```

   Now let's say we often are lazy, and tend to type proper names in all lower case. We'd like all hotel names to use title case (we just don't care for those new-fangled names beginning with lower case letters), but instead of adding a validation error, we want to just have our model fix the case of the name.

5. We can achieve this using a callback:

   ```
   class Hotel < ActiveRecord::Base
     validates :name, :presence => true
     validates_length_of :name, :minimum => 2


     before_save :ensure_name_is_in_titlecase


     protected
   ```

```
  def ensure_name_is_in_titlecase
    self.name = self.name.titlecase
  end
end
```

6. Go to the console and test the callback:

   ```
   $ rails c

   > hotel = Hotel.new(name: "the ee cummings inn")
   > hotel.save
   > hotel.name
    => "The Ee Cummings Inn"
   ```

7. Try experimenting with some other callbacks, such as `after_save`, and `after_create`.

Associations

Association Basics

Associations define relationships between models. Unlike column names mapping to instance methods, Rails cannot infer relationships from foreign keys in the database, so you must link objects explicitly. Associations allow Rails to provide many convenient methods to refer to dependent objects and vice versa. There are four main associations with two additional variations:

```
belongs_to
has_one
has_many
has_many :through
has_one :through
has_and_belongs_to_many
```

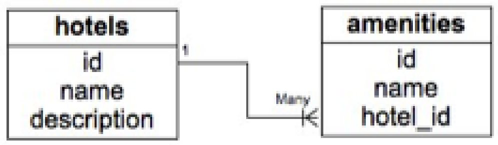
- The `belongs_to` association tells us that this object relates to a parent object. Generally, the model that contains a belongs_to will physically hold a foreign key referencing that object, which must be defined in a migration unless it already exists on the table definition.

- The `has_one` association tells ActiveRecord that there is one, and only one, related model element to this element.

- The `has_many` association defines a reference to a child table. The model defining this will have a collection with the name of the element. You may define a corresponding reverse `belongs_to` relationship in the child to establish a reverse relationship. As with the `belongs_to` relationship, the child table must contain a foreign key definition (in this case it is the table pointed to by the child model).

  Suppose we wish to keep track of amenities at a hotel in a separate table, assuming that a hotel can have many amenities but that each amenity is unique to a single hotel.

Example



The file `app/models/hotel.rb` contains:

```
class Hotel < ActiveRecord::Base
  has_many :amenities
end
```

The file `app/models/amenity.rb` contains:

```
class Amenity < ActiveRecord::Base
  belongs_to :hotel
end
```

An example of usage might be:

```
@hotel = Hotel.find(21)
@amenity = @hotel.amenities.create(name: "Sunflower Spa")
```

Associations Tips

The model that corresponds to the table with the foreign key column `belongs_to` the model that is referred to by the foreign key.

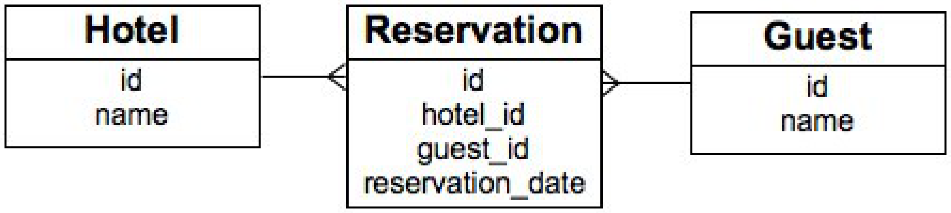
Many-to-Many Relationships

Many developers primarily use `has_many :through` associations for many-to-many relationships, for the benefit of utilizing the intermediate model with the ability to have extra attributes. This is in contrast to `has_and_belongs_to_many` associations. This association indicates that the declaring model can be matched by zero or more instances of a model by proceeding through a third model (commonly referred to as the join table in many-to-many relationships). For example:

```
class Hotel < ActiveRecord::Base
  has_many :reservations
  has_many :guests, :through => :reservations
end

 class Reservation < ActiveRecord::Base
   belongs_to :hotel
   belongs_to :guest
 end

 class Guest < ActiveRecord::Base
   has_many :reservations
   has_many :hotels, :through => :reservations
 end
```



In Rails 3.1+, you also have the ability to have *nested* `has_many :through` associations, giving you the ability to have something like this:

```
class Hotel < ActiveRecord::Base
  has_many :reservations
  has_many :guests, :through => :reservations
  has_many :airports, :through => :guests
end
```

Walkthrough 4-5: Creating Associations

In this walkthrough you will create associations between the Hotel model and the Cities model.

Steps

This lab depends on the city model, which is created in Walkthrough 4-1.

1. Add the association to the Hotel model, contained in `app/models/hotel.rb`

```
class Hotel < ActiveRecord::Base
  validates :name, :presence => true
  validates_length_of :name, :minimum => 2


  before_save :ensure_name_is_in_titlecase


  belongs_to :city


  def ensure_name_is_in_titlecase
    self.name = self.name.titlecase
  end
end
```

2. Test adding a city to a hotel instance in the Rails console:

```
$ rails c

hotel = Hotel.new
hotel.city = City.find(1)
hotel.city
# should display a hash representing the properties of the hotel.city
```

3. Add the association to `city.rb`.

```
class City < ActiveRecord::Base
  has_many :hotels
end
```

4. Verify that the City model can fetch related hotels:

   ```
   $ rails c
   ```

   ```
   > city = City.new("Fort Washington")
   > hotel = Hotel.new(:city => city, :name => "Blue Hotel")
   > city.hotels
   # should return an Array including the newly minted Hotel
   ```

5. Write a migration to add a foreign key column to hotels.

   ```
   $ rails generate migration AddCityReferenceToHotels
   ```

6. Replace the migrations in this file with a single `change` method. The `change` method can handle any of the well-known migrations by running the positive version, such as `add_column` during a forward migration, and the inverse of this method, 'remove_column', during a rollback.

   ```
   def change
     add_column :hotels, :city_id, :integer
   end
   ```

7. Run the migration.

Active Record Queries

Finder Methods

ActiveRecord includes a number of chain-able finder methods to construct queries.

- where
- select
- group
- order
- limit
- offset
- joins
- includes
- lock
- readonly
- from

The finder methods return instances of `ActiveRecord::Relation`, which respond to not only finder methods but also ActiveRecord methods.

```
luxury_hotels = Hotel.where(:star_rating => 5)
new_luxury_hotel = luxury_hotels.new
new_luxury_hotel.star_rating # should return 5
```

Lazy Loading

Relations do not query the database until necessary, i.e. when an enumerable method is called.

Scopes

Prior to Rails 3, named scopes were the best way to create some of the functionality of the Relation.

```
class Hotel < ActiveRecord::Base
    ...
    named_scope :luxury, :conditions => { :star_rating => 5 }
end

# inside controller method, for example
luxury_hotels = Hotel.luxury
```

Rails 3 preserves the functionality of `named_scope` with the new keyword, `scope`.

```
scope :luxury, where(:star_rating => 5)
```

However, because relations are lazy loaded, class methods are an equally efficient, cleaner, more idiomatic way to achieve the same result.

```
class Hotel < ActiveRecord::Base
  ...
  def self.luxury
    where(:star_rating => 5)
  end
end
```

NOTE: This also allows you to create dynamic scopes without having to use lambdas.

Walkthrough 4-6: Creating a Query

In this walkthrough, you will add a `star_rating` property to hotels and create a class method to return a list of hotels in the system with a specific rating.

Steps

1. Write a migration to add the `star_rating` property to hotels.

   ```
   $ rails generate migration add_star_rating_to_hotels
   ```

2. To the generated migration file add the new column to the up and down methods.

   ```
   def self.up
     add_column :hotels, :star_rating, :integer
   end


   def self.down
     remove_column :hotels, :star_rating
   end
   ```

3. Run the migration:

```
rake db:migrate
```

4. Set up seeds for several hotels in `db/seeds.rb`

```
Hotel.create([
  { :name => "Hollywood Hotel", :description => "the one and only",
    :star_rating => 1 },
  { :name => "200th Hotel", :description => "Why not rate me?"},
  { :name => "Bates Motel",
    :description => "This is a very long description.  This comment  has " +
    " enough of the description to see '...'",
    :star_rating => 3 }
])
```

5. Let's re-initialize our database by dropping it, and running another rake command, `db:setup`, which creates the database, executes the database setup code in `db/schema.rb`, and re-runs any code in `seeds.rb`:

```
rake db:drop
rake db:setup
```

   You'll want to use this whenever you set up a fresh environment, or if you re-do your seed data. If you are introducing a new feature, you can also write a data modification script that you can execute via `rails runner script.rb`.

6. Write the class method `rated` in `app/models/hotel.rb` so that you can test it in the console.

```
class Hotel < ActiveRecord::Base
  ...


  def self.rated(star_rating = 3)
    where("star_rating >= ?", star_rating)
  end


  protected
  ...
end
```

7. Execute the `rated` call:

```
$ rails c
> Hotel.rated(2)
```

8. Experiment by chaining this with other ActiveRecord query methods:

```
> Hotel.rated(2).sort(:name)
```

Unit Review

- ORM
- ActiveRecord
- Databases
- Migrations
- DB Console
- Models
- Console
- Seeds
- Validations
- Associations
- Queries

Lab 4: Writing model methods

In this lab you will create the Trip model with a validation and associate it to the existing Hotel model from the walkthroughs.

Objectives

After completing this lab you should be able to:

- Write migrations to create database tables
- Create, edit, and save models from the rails console
- Write validations for models
- Write associations for models
- Access the database using rails' database console

Steps

1. Complete the steps from the walkthroughs if you have not already so that you have a working Hotel model.

2. Create the migration for the Trip model with two attributes, name and date.

3. Run the migration.

4. Check the database with the Rails DB console to verify creation of table.

5. Use `rake db:rollback` to undo the migration. Ensure that the trips table is no longer in the database. Recreate the table using the `rake db:migrate` task.

6. Create the model class definition, `app/models/trip.rb`.

7. Create and save the new object in rails console.

8. Add validation to `trip.rb` - require that the name is unique and that it cannot be empty.

9. Using the `create` method on the Trip class, create an invalid object in the console.

10. Try to save the invalid trip.

11. Check to see whether the trip is valid by calling the `valid?` method. Using the `errors` property, check the reason for the error.

12. Fix the invalid trip.

13. Check the validity of the trip again. If it is still invalid, modify it until it passes validation.

14. Save the valid object.

15. Using the model files `hotel.rb` and `trip.rb`, add a bidirectional association between Trip and Hotel with the following rules: Hotels can be visited many times. Trips are to a given hotel.

16. Create a migration using `rails g migration` to add the foreign key column to the appropriate model object. The type required will be `integer`. If you use the special syntax `rails generate migration add_fieldname_to_tablename fieldname:datatype`, rails will not only generate a stub file, it will also generate the actual change method *in* the migration as well. Go ahead, try the following command and inspect the generated file:

    ```
    rails generate migration add_hotel_id_to_trips hotel_id:integer
    ```

17. Run the migration.

18. Using the `rails console` command, create several trips and associate them with a hotel. Then, retrieve the Hotel and check to see that the `visits` association column contains associations.

19. Generate and inspect `db/schema.rb`.

20. Edit `db/seeds.rb` to populate database with five visits to hotels, and issue the `rake db:seed` command. Verify that the visits were persisted by entering queries such as `Visit.all` and `Hotel.find_all_where_visit_id(_number_)` in the Rails console.

Unit 5: Views

Unit Objectives

After completing this unit you should be able to:

- Use ERB to create views
- ERB versus HAML
- Use layouts and helpers to DRY up your views
- Understanding the Rails asset pipeline

Unit Topics

- ERB
- HAML
- Views
- Layouts
- Helpers
- Asset pipeline

Views

View Basics

Rails views are HTML templates which are rendered when a controller processes an action. A simple view is rendered all by itself, but it can also be wrapped in a layout and contain partial views. If a controller does not have a method defined for a particular action, it will simply look for the corresponding view template and render it. By default, views are located in a directory named after the corresponding model under `app/views/`.

View filenames are constructed of the action name, followed by the response format, followed by the template language. A standard index view would be named `index.html.erb`.

Basic ERB Syntax

The default language for view templates in Rails is ERB, a templating system built in to Ruby. ERB uses a set of tags, similar to PHP or JSP, that can be interspersed with static HTML.

```
<% Ruby code (not displayed) %>
<%= Ruby expression (displayed) %>
<%# comment %>
```

ERB v. Haml

ERB files consist of standard HTML and Ruby code wrapped in ERB tags. For developers accustomed to and comfortable with HTML, ERB is familiar and easy to work with. There are many alternative templating languages. Haml is one of the most popular, offering a cleaner more succinct syntax that some find easier to parse visually and efficient to write. Haml's main distinctions are its omission of the brackets used in HTML and ERB and its use of white space to handle nesting (with all of the same controversies that Python engenders for its use of white space).

The downside to Haml is that it is not very friendly for web or graphic designers who are not familiar with the different syntax. It adds a learning curve that might not be budgeted in your project, so some prefer to stick with ERB as it nicely fits into both the Ruby and HTML world.

In this class we will use ERB since it is the default rendering engine for Rails, but we will discuss Haml as well so you can make the choice for yourself.

Haml Basics

Haml uses two spaces to indicate nesting, as a result closing tags (for markup) and end tags (for code) are not required.

ERB

```
<div id="profile">
  <div class="left column">
    <div id="date"><%= print_date %></div>
    <div id="address"><%= current_user.address %></div>
  </div>
    <div class="right column">
    <div id="email"><%= current_user.email %></div>
    <div id="bio"><%= current_user.bio %></div>
  </div>
</div>
```

Haml

```
#profile
  .left.column
    #date= print_date
    #address= current_user.address
  .column
    #email= current_user.email
    #bio= current_user.bio
```

HTML entities are prefixed with %, and HTML attributes are expressed as Ruby hashes. So, for example:

HTML:

```
<table border="0" cellspacing="5">
```

HAML:

```
%table{:border => 0, :cellspacing => 5}
```

Classes and IDs are expressed using the same syntax as CSS. If you omit the HTML entitiy when specifying class and/or ID information, Haml creates a `div` by default.

```
<div id="navigation" class="toolbar">
```

```
#navigation.toolbar
```

Just as in ERB, = followed by a Ruby expression will be evaluated and output. Use - before Ruby code that has no output. Use -# for Haml comments.

```
-# A simple loop
%ul
  - @hotels.each do |hotel|
    %li= hotel.name
```

Haml Gotchas

The most common Haml problems have to do with whitespace. The indent for each level of nesting must be exactly two spaces. Tabs are not allowed. Set your text editor to use "soft tabs" of exactly two spaces, and when copying and pasting, be sure your nesting is correct. The other main issue with Haml is that a single statement cannot contain any line breaks. This is intentional on the part of Haml to discourage large chunks of code in the view. There are two exceptions to the line break restriction. You can use the pipe (|) to break multiline strings, and a line break is permitted immediately after commas between attributes of an entity.

From the Haml documentation:

```
%whoo
  %hoo= h(                          |
    "I think this might get " +     |
    "pretty long so I should " +    |
    "probably make it " +           |
    "multiline so it doesn't " +    |
    "look awful.")                  |
  %p This is short.
```

and

```
%script{:type => "text/javascript",
        :src  => "javascripts/script_#{2 + 7}"}
```

Walkthrough 5-1: Creating a View

In this walkthrough, you will create an index view for hotels.

Steps

1. Before we get into rendering a list of hotels using an ERB template, let's format the list of hotels in a terminal window using a Ruby script. Paste the following code, which loops through the hotels and prints out the hotel name and description (separated by a tab via `\t`), into a new file, `lib/scripts/hotel_formatter.rb`:

   ```
   Hotel.all.each do |hotel|
     puts "#{hotel.name}\t#{hotel.description}"
   end
   ```

2. You can use the Rails runner to run the script with full access to your Rails application's environment. The runner enables you to write scripts that reference Rails models without needing to add code that loads those classes. Run the hotel formatting script by entering:

   ```
   rails r lib/scripts/hotel_formatter.rb
   ```

   The `r` is a shortcut for `runner`.

3. The next step is to execute this script when the ERB template for the Hotels index page is accessed from the browser. Paste the script we used in Step 1 into a new file, `app/views/hotels/index.html.erb`:

   ```
   Hotel.all.each do |hotel|
     puts "#{hotel.name}\t#{hotel.description}"
   end
   ```

   Next use the `<%` and `%>` ERB tags to specify code that should be executed but not displayed as follows:

   ```
   <% Hotel.all.each do |hotel| %>
     puts "#{hotel.name}\t#{hotel.description}"
   <% end %>
   ```

   Finally, remove the `puts` method call and use the `<%=` and `%>` to indicate that the hotel name and description should be displayed as follows:

```
<% Hotel.all.each do |hotel| %>
  <%= "#{hotel.name}\t#{hotel.description}" %>
<% end %>
```

4. In order to see hotels index page, we'll need to add routes for hotels to the routes configuration file and we'll need to generate a hotels controller.

   Insert the following line inside the main block in `config/routes.rb`. We'll explore routes in more detail in Unit 6.

   ```
   resources :hotels
   ```

   Generate the hotels controller using `rails g controller hotels`. This command will generate several files and directories in anticipation that you'll be using them in conjunction with the hotels controller. It would generate the `app/views/hotels` folder if we had not already created it.

   Take a look at the generated controller. As you'll see there is no logic in the file. You are about to experience some of what is often referred to as "Rails magic." With 0 lines of code in the controller, the Rails framework will know how to handle the URL: http://localhost:3000/hotels.

5. View the page in your browser (http://localhost:3000/hotels). You'll notice that all the names and descriptions run together on one line. To fix this, we'll start mixing HTML into the Ruby code in `app/views/hotels/index.html.erb`.

6. Add an HTML line break tag (`<br>/`) following the line that prints each hotel's name and description as follows:

   ```
   <% Hotel.all.each do |hotel| %>
     <%=  "#{hotel.name}\t#{hotel.description}" %>
     <br/>
   <% end %>
   ```

   Access the page in a browser to verify that the fix worked.

7. Now modify the page to use HTML table columns to separate a record's database column values instead of tabs as follows:

   ```
   <table>
   <% Hotel.all.each do |hotel| %>
       <tr>
         <td><%= hotel.name %></td>
         <td><%= hotel.description %></td>
       </tr>
   <% end %>
   </table>
   ```

8. Add HTML table headers as follows:

```
<table>
  <tr>
    <th>Name</td>
    <th>Description</td>
  <tr>
<% Hotel.all.each do |hotel| %>
  <tr>
    <td><%= hotel.name %></td>
    <td><%= hotel.description %></td>
  </tr>
<% end %>
</table>
```

9. View the page in a browser.

Layouts

Layout Basics

Exemplifying the principle of "Don't Repeat Yourself" which suffuses the design of Rails, layouts in Rails simplify the use of markup that is reused on multiple pages, such as HTML title and head blocks as well as application headers and footers.

Rails uses the default application layout, in our case `app/views/layouts/application.html.erb` since we are using Haml, unless it finds a layout specific to the model for the view being rendered. Layouts can also be disabled or overridden explicitly in the controller.

A simple layout might look as follows:

```
<html>
<head>
  <title>Traveljournal</title>
</head>
<body>
  <%= yield %>
</body>
</html>
```

The page template being rendered will be inserted in place of the `yield` method. The standard Rails layout template includes a few additional items.

```
<!DOCTYPE html>
<html>
<head>
  <title>Traveljournal</title>
  <%= stylesheet_link_tag    "application" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

The `stylesheet_include_tag` and `javascript_include_tag` statements include the Rails default javascript files and stylesheets. The settings for these defaults can be customized or individual stylesheet and javascript files can be specified. Finally, the `csrf_meta_tag` statement provides protection against cross-site scripting attacks.

Walkthrough 5-2: Editing a layout

In this walkthrough, you will edit an application-wide layout.

Steps

1. Open `app/views/layouts/application.html.erb`

2. The `rails new` generated this template when we created the traveljournal application. Let's edit it to make the title appear nicer and give pages across the application a common banner.

3. Replace the generated title (ie "`traveljournal`") with the more presentably formatted string "Travel Journal":

   `**<title>Travel Journal</title>**`

4. Add a banner on the next line after the opening `body` tag, and sandwich the `yield` line with horizontal rules so the body looks as follows:

   ```
       <body>
         <h1 class="banner">Travel Journal</h1>
         <hr class="rule"/>
         <%= yield %>
         <hr class="rule"/>
       </body>
   ```

5. Finally add these site-wide CSS directives. Rather than straight CSS, you're going to use SASS, a CSS pre-processing tool that helps simplify style sheet syntax and reduces repetition.

   SASS has two file formats, the older `.css.sass` format, and the newer `.css.scss` format. We'll use the newer format.

   Place the styles for the "banner" and "rule" classes in a new file, `app/assets/stylesheets/common.css.scss`:

   ```
   $base-color: red;


   .banner {
     font: {
       style: italic;
       weight: bold;
      }
     color: $base-color;
   }


   .rule {
     height: 2px;
   ```

```
  border: none;
  background-color: $base-color;
  color: $base-color;
}
```

This file will be detected and run through a processing step, and Rails will emit normal CSS to the browser.

6. Experiment with various settings in the SCSS file. Change the `$base-color` attribute to `blue` or `green`, for example. When you re-load the browser, your style will change.

7. If you want to see how easy it is to use traditional CSS instead of SASS/SCSS, you can rename `common.css.scss` to `common.css` (i.e. remove the "scss" suffix) and replace the SCSS code with following CSS. Save it, and then refresh your browser:

```
.banner {
  font-style: italic;
  font-weight: bold;
  color: red;
}


.rule {
  height: 2px;
  border: none;
  background-color: red;
  color: red;
}
```

Helpers

Helper Basics

View helpers are another way to avoid repetition in your application. Any display or formatting logic that occurs in your views can be extracted into helpers. Each model has a corresponding view helper located in `/app/helpers/`.

Quite simply any logic in your view can be made into a method in the view helper, which then can easily be reused, or simply used to make your views cleaner and more readable.

For example, suppose we want to convert each hotels star rating to a string of asterisks to display next to the hotel name on the index page.

```
def display_stars(star_rating=nil)
  if star_rating
    stars_str = ""
    star_rating.times do
      stars_str.concat("*")
    end
    stars_str
  else
    "Not Rated"
  end
end
```

Take note that helpers you create in any file will be included and available to all of your views. If you wish to only include the helpers in `application_helper.rb` and your controller's helper file, then call the `clear_helpers` method inside your ApplicationController.

```
class ApplicationController < ActionController::Base
  clear_helpers
  protect_from_forgery
end
```

Walkthrough 5-3: Creating a Helper

In this walkthrough, you will create a simple view helper that displays formats today's date using `Time#strftime`.

Steps

1. Use the Time#strftime documentation (http://www.ruby-doc.org/core/classes/Time.html#M000392) to help you format `Date.current` a few different ways in the Rails console. Try using `Time#strftime` to print out the day of the week. Try to match `Monday, January 1, 2011`. Now try to match `01/01/2011`, which is the format we'll be using for our helper.

2. Paste the following method definition into `app/helpers/application_helper.rb`. Even though helper methods defined in `app/helpers/hotels_helper.rb` (or a helper file paired with any other model's controller) are accessible to any `traveljournal` controller's views, it's good practice to put methods that are likely to be used for more than one controller's views in `app/helpers/application_helper.rb`.

   ```
   module ApplicationHelper
     def formatted_current_date
       Date.today.strftime("%m/%d/%Y")
     end
   end
   ```

3. Use the helper to format the date in the phrase "Hotels as of [today's date goes here]" on the Hotels index page (`app/views/hotels/index.html.erb`) as follows:

   ```
   <h2>Hotels as of <%= formatted_current_date %></h2>
   <table>
     <tr>
       <th>Name</th>
       <th>Header</th>
     <tr>
   <% Hotel.all.each do |hotel| %>
     <tr>
       <td><%= hotel.name %></td>
       <td><%= hotel.description %></td>
     </tr>
   <% end %>
   ```

4. View the results in your web browser at `http://localhost:3000/hotels`.

Lab 5: Creating views, layouts, & helpers

Objectives

After completing this lab you should be able to:

- Create views using ERB
- Create layouts
- Create view helpers

Steps

Create an index view

1. Generate a controller for trips using `rails g controller trips`.

2. Update `config/routes.rb` to include routes for trips. Add the following directive (which we will discuss in the next chapter):

   ```
   resources :trips
   ```

3. Create an index view for trips similar to the view we created for hotels, including a "Trips as of" header that uses the `formatted_current_date` helper. For each trip record, display the trip name and date.

4. View the page in your browser at `http://localhost:3000/trips`.

5. One problem with this page is that the date format in the Trips table (i.e. the default formatting for `Trip#date`) is not consistent with the format in the "Trips as of" banner. We'll need to write a helper similar to `formatted_current_date` in order to format the trip dates. We'll need a method that is not hard-wired to only format today's date.

   Edit `app/helpers/application_helper.rb` to add the following method, which accepts the date to format as an argument:

   ```
   def formatted_date(date)
     date.strftime("%m/%d/%Y")
   end
   ```

   Reduce code duplication in `app/helpers/application_helper.rb` by refactoring `formatted_current_date` to use the new more generic `formatted_date` as follows:

   ```
   def formatted_current_date
     formatted_date(Date.current)
   end
   ```

   Use the new `formatted_date` helper to format the trip dates to be consistent with the "Trips as of" banner.

View the page in your browser.

Create some helpers

1. Add a helper that truncates long hotel descriptions but displays the whole description, tooltip-style, when the user mouses over the description. For the purposes of this exercise, a description more than 25 characters in length is considered long.

   The `description_column` helper below leverages the Rails text helper `truncate` (`http://api.rubyonrails.org/classes/ActionView/Helpers/TextHelper.html#method-i-truncate`). If the string passed to `truncate` as the first argument has more characters than the specified `:length`, it returns a version of the string truncated at the specified length and suffixed with `...` (the developer has option of specifying a character in lieu of `...` to indicate that text was omitted).

   The tooltip-style behavior is implemented via the HTML `title` attribute.

   The `html_safe` call prevents Rails from escaping the HTML. As a security measure, Rails assumes that text that will be output to a view is not safe.

   Edit `app/helpers/hotels_helper.rb` to add the description_column helper. This helper is specific to the hotels table. It differs from the other helpers we've added so far because it returns HTML output instead of a text string. ** Note, please remove the existing `<td>...</td>` entry as otherwise you'll get a nested `<td>` pair.

```
def description_column(description)
  "<td title='#{description}'>#{truncate(description,:length => 25)}</td>".html_safe
end
```

   Replace the description `td` element in `app/views/hotels/index.html.erb` with code that calls `description_column` and passes it the hotel description. With this change, your table should look like this:

```
<table>
  <tr>
    <th>Name</td>
    <th>Description</td>
  </tr>
    <% Hotel.all.each do |hotel| %>
      <tr>
        <td><%= hotel.name %></td>
        <%= description_column(hotel.description) %>
      </tr>
    <% end %>
</table>
```

   The helper works fine, but could be refactored to make it easier to maintain. Rails provides helpers that make it easier to create helpers that generate HTML output. You can use `content_tag` to make `description_column` less unwieldy.

   The Rails `content_tag` helper takes the following parameters:

   - a type of HTML element (in this case, `:td`),
   - the content for the element (in this case "`#{truncate(description,:length => 25)}`")

- and HTML attributes that should be assigned to the HTML element in the form of hash keys and values (in this case, `:title=>desc`)

The helper then returns the code for the HTML element. You can read more about `content_tag` here:
http://api.rubyonrails.org/classes/ActionView/Helpers/TagHelper.html#method-i-content_tag

Replace the description_column definition in `app/helpers/application_helper.rb` with the following improved version and then verify the the hotels index still displays properly.

```
def description_column(description)
  content_tag(:td,truncate(description,:length => 25),:title=>description)
end
```

2. Create a helper called `as_of_banner` that can be used by both the hotels index and the trips index to display the "as of" banner. The helper should live in `app/helpers/application_helper.rb` and should take the model name as an argument.

   Modify hotels index view and the trips index view to use the new `as_of_banner` helper.

   Verify that the new helper works by viewing both screens in your browser.

Take HAML for a test drive

1. To get a taste of Haml, we'll rewrite the hotels index view using Haml.

   Haml is not packaged with Rails by default, so add the line `gem 'haml'` to the `Gemfile` file in the `traveljournal` directory and run `bundle install`.

   Rename `app/views/hotels/index.html.erb` to `app/views/hotels/do_not_use.index.html.erb`. By default, Rails will looks for an erb template prefixed with the action name if the controller does not provide explicit alternative directions. If a file meeting this criteria is not found, Rails will look to see if there are other templates that match the action name (in this case, `index`) in the views directory paired with the controller.

   Paste the following Haml version of the hotels index view into a new file, `app/views/hotels/index.html.haml` and verify that the hotels index displays properly.

```
%h2= as_of_banner("Hotels")


%table
  %tr
    %th Name
    %th Description


  - Hotel.all.each do |hotel|
    %tr
      %td= hotel.name
      %td= hotel.description
```

2. Rewrite the trips index page using Haml and verify that the page displays correctly in your browser. Remember to rename the existing `/app/views/trips/index.html.erb` file to `do_not_use.index.html.erb` so that Rails will serve the Haml version.

3. As a last step, revert to the name `index.html.erb` from `do_not_use.index.html.erb`, which will restore the original ERB file. When both index views are present, Rails will use the ERB template by default. Run it again and make sure the view renders. You may experiment with content in the ERB to prove that it is being rendered.

Unit 6: Controllers

Unit Objectives

After completing this unit you should be able to:

- Understand the relationship between CRUD and REST
- Populate Views with Model data using Controllers
- Use ActionController filters
- Respond to HTML, XML, and JSON requests

Unit Topics

- CRUD & REST
- Routing
- Controllers
- Filters

CRUD & REST

The CRUD acronym describes the four basic functions of most database-backed web applications (and persistent storage in general):

- Create
- Update
- Read
- Delete

These four operations map neatly onto the four HTTP verbs:

- POST
- PUT
- GET
- DELETE

REST is almost an acronym which stands for Representational State Transfer, an architecture designed by Roy Fielding (one of the authors of the HTTP protocol) for scalability, generality, and encapsulation of requests, particularly over HTTP.

A RESTful architecture consists of: separation of concerns between client and server, statelessness, cacheability, indifference to layers, extensibility on demand, and a uniform interface between clients and servers.

A key concept in REST is that the basic unit, resources, are distinct from the representations of those resources. A request for a specific resource, for example, can receive a response in a variety of formats and languages, depending on the request context.

Another key concept is that GET, PUT, and DELETE are idempotent.

Routing

RESTful Defaults

Rails implements elements of REST mapping CRUD operations onto controller actions. A default Rails route contains seven actions:

```
GET    /hotels(.:format)
{:action=>"index", :controller=>"hotels"}

POST   /hotels(.:format)
{:action=>"create", :controller=>"hotels"}

GET    /hotels/new(.:format)
{:action=>"new", :controller=>"hotels"}

GET    /hotels/:id/edit(.:format)
{:action=>"edit", :controller=>"hotels"}

GET    /hotels/:id(.:format)
{:action=>"show", :controller=>"hotels"}

PUT    /hotels/:id(.:format)
{:action=>"update", :controller=>"hotels"}

DELETE /hotels/:id(.:format)
{:action=>"destroy", :controller=>"hotels"}
```

Examples:

```
http://traveljournal.com/hotels
http://traveljournal.com/hotels/12
http://traveljournal.com/hotels/12.xml
http://traveljournal.com/hotels/12.json
http://traveljournal.com/hotels/new
http://traveljournal.com/hotels/12/edit
```

Walkthrough 6-1: Creating Routes

In this walkthrough you will create a simple route for the Hotel model.

Steps

1. Go to the front page ("/") and you'll notice that we see the default Rails page.

2. Add root route

   ```
   root :to => "hotels#index"
   ```

3. Delete `public/index.html`.

4. Restart the the app. The front page should now show our list of hotels.

5. Have the root action redirect to `/hotels` instead of displaying the page directly.

   ```
   #root :to => "hotels#index"
   root :to => redirect("/hotels")
   ```

6. Restart server and try it out.

   Rails has a feature-rich routing mechanism that allows for user-friendly, restful URLs, support for subdomains, and nested pathing such as `/order/1/item/13`. For more information, refer to the well-commented sample routes in `routes.rb`, or consult the rails guide, Routing from the Outside In, at http://guides.rubyonrails.org/routing.html.

Controllers

Controllers are subclassed from ApplicationController which is in turn subclassed from ActionController::Base. A standard RESTful controller contains seven actions, index, show, new, edit, create, update, destroy. Controllers do not have to conform to this RESTful model and can contain additional actions or none of these actions. A standard route assumes these seven actions, but you can easily write custom routes to handle any type of action in a controller.

The controller handles the interaction between the view and the model by receiving a request from the browser (or other user agent), sometimes with query parameters, interacting with the model as needed, then rendering the view in the format requested, often HTML but also XML, JSON, or a custom format.

A standard controller, generated from the rails scaffolding command `rails g scaffold Hotels`, in its simplest form:

```
class HotelsController < ApplicationController
  def index
    @hotels = Hotel.all
  end

  def show
    @hotel = Hotel.find(params[:id])
  end

  def new
    @hotel = Hotel.new
  end

  def edit
    @hotel = Hotel.find(params[:id])
  end

  def create
    @hotel = Hotel.new(params[:hotel])

    respond_to do |format|
      if @hotel.save
        redirect_to(@hotel, :notice => 'Hotel was successfully created.')
      else
        render :action => "new"
      end
```

```
      end
    end

    def update
      @hotel = Hotel.find(params[:id])

      respond_to do |format|
        if @hotel.update_attributes(params[:hotel])
          redirect_to(@hotel, :notice => 'Hotel was successfully updated.')
        else
          render :action => "edit"
        end
      end
    end

    def destroy
      @hotel = Hotel.find(params[:id])
      @hotel.destroy

      respond_to do |format|
        redirect_to(hotels_url)
      end
    end
  end
```

Rendering

A controller action will try to render a corresponding view template by default, which is why you don't need to include `render` in the following:

```
def edit
  @hotel = Hotel.find(params[:id])
end
```

However, you have complete control over constructing the response if you need to do something special:

```
# render a different action
render :action => 'new'

# render a template
render :template => 'shared/something'

# render a file
render :file => '/tmp/random_file.erb'

# render text
render :text => 'done'
```

Unified Rendering

As of Rails 2.3, the above can be simplified further:

```
render 'new'
render :new
render 'shared/something'
render '/tmp/random_file.erb'
```

Filters

Filters are methods that can be run before or after – or, less commonly, around – controller actions (`before_filter`, `after_filter`, `around_filter`). Filters can be applied to specific actions using `:only` or `:except`.

For example:

```
before_filter :foo
before_filter :bar, :only => :index
before_filter :fnord, :except => [:new, :edit]
```

Any filters specified in ApplicationController will be inherited by other controllers. There are two ways to prevent ApplicationController filters from running in other controllers. You can either a) create subclasses of ApplicationController that contain your filters and subclass those controllers instead of ApplicationController, or you can b) explicitly skip specific filters. Authentication is a common example where filters need to be skipped.

In the first case:

```
class ApplicationController < ActionController::Base
end

class AuthenticatedController < ApplicationController
  before_filter :authenticate_user
end

class SessionController < ApplicationController
end

class HotelsController < AuthenticatedController
end
```

In the second case:

```
class ApplicationController < ActionController::Base
  before_filter :authenticate_user
end

class SessionController < ApplicationController
  skip_before_filter :authenticate_user
end

class HotelsController < ApplicationController
end
```

Walkthrough 6-2: Adding Controller Methods

Steps

1. Open `app/controllers/hotels_controller.rb`.

2. Add an index method that will load our Hotel objects.

```
class HotelsController < ApplicationController
  def index
    @hotels = Hotel.order(:name)
  end
end
```

3. Update `app/views/hotels/index.html.erb` to use the instance variable `@hotels` rather than performing the query from the view itself (always a bad idea, as you never want to embed persistence logic within a view). This is literally a one line change - replace `Hotel.all.each` with `@hotels.each`:

```
<table>
  <tr>
    <th>Name</td>
    <th>Description</td>
  </tr>
    <% @hotels.each do |hotel| %>  <-- change this line
      <tr>
        <td><%= hotel.name %></td>
        <td><%= description_column(hotel.description) %></td>
      </tr>
    <% end %>
</table>
```

4. View the results in your browser by browsing to `http://localhost:3000` (remember, we made the `hotels#index` action the root of the web application).

5. Modify the controller to add a filter, `load_hotel` which loads the proper hotel object based on the `:id` passed in via the URL path, where a sample URL is /hotels/123, or /hotels/123/edit.

```
class HotelsController < ApplicationController
  before_filter :load_hotel, :except => [:new, :index, :create]


  # leave the index method as-is


  protected


  def load_hotel
    @hotel = Hotel.find(params[:id])
  end
end
```

This filter picks up the id from the request URL and uses it to load the instance variable `@hotel`. This is useful for the `show`, `edit`, and `destroy` methods, so that they do not have to repeat the same fetch logic.

6. Add an empty `show` method:

```
def show
end
```

The `:new, :index,` and `:create` actions do not need this filter as they handle the hotel object differently.

7. Since we have added the `show` action, let's create a corresponding view to display the individual hotel. Open `app/views/hotels/show.html.erb`:

```
<h1><%= @hotel.name %></h1>
<p><%= @hotel.description %></p>
```

8. Now back in the `app/views/hotels/index.html.erb` code, we can create a link to the show action for each hotel. Add a third column header to the table header row, and a third data column to provide the link.

```
<h1>Hotels as of <%= day_today %></h1>
<table>
  <tr>
    ...
    <th>Actions...</th>
    <% @hotels.each do |hotel| %>
      <tr>
        ...
       <td><%= link_to hotel.name, hotel_path(hotel) %></td>
        ...
        <td><%= link_to "Edit", edit_hotel_path(hotel) %>
            <%= link_to 'Destroy', hotel,
                confirm: 'Are you sure?', method: :delete %></td>
        </td>
      </tr>
     <% end %>
```

The `link_to` helper accepts the link text as the first argument, and the path as the second. You can use either a full URL, the `_path` or `_url` helper methods, or if you pass a specific model instance it will go to the show action.

9. The `resources :hotels` route automatically provides helpful paths, such as `edit_hotel_path`, that can provide proper URLs to various actions, such as `/hotels/123/edit`. You can see all of these with the `rake routes` command. Try it now.

Walkthrough 6-3 - Response Formats

In this walkthrough you will modify `hotels_controller.rb` to take advantage of Rails' support for XML and JSON requests.

Steps

1. In `app/controllers/hotels_controller.rb`, add the `respond_to` method on line 2 after the class definition and modify the `index` method:

```
class HotelsController < ApplicationController
  respond_to :html, :json
  before_filter :load_hotel, :except => [:new, :index, :create]


  def index
  @hotels = Hotel.order(:name)
    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @hotels }
      end
    end
    ...
  end
```

2. You can use the browser to view the JSON response by appending .json to the URL.

   ```
   http://localhost:3000/hotels.json
   ```

3. You can add XML support simply by adding `:xml` after `:xml` to line 2 of the controller.

   ```
   respond_to :html, :xml, :json
   ```

4. You can use the browser to view the XML by appending .xml to the URL.

   ```
   http://localhost:3000/hotels.xml
   ```

Unit Review

- CRUD
- REST
- Routing
- Seven RESTful actions
- Controllers
- Filters
- Content Conversion

Lab 6: Connecting Models and Views using Controllers

Objectives

In this lab you will

- Create routes to structure application URLs
- Write controller methods to load live model data into views
- Write a before filter to execute common code and DRY up the controller.

Steps

1. Write `trips#index` method.
2. Write `trips#show` method.
3. Add a `before_filter` filter to the trips controller.
4. Add some trips to the database.
5. Link each trip in the index to its corresponding show page.
6. View in browser.

Unit 7: Forms

Unit Objectives

After completing this unit you should be able to:

- Use forms to create, update and delete records
- Take advantage of form helpers and custom form builders to simplify view code.

Unit Topics

- Forms
- Form Helpers
- Custom Form Builders
- Nested Forms

Forms

Form Helpers

Rails provides two sets of helpers for simplifying the tedium of managing form tags and form fields. One set is generic and assumes nothing about the form; the other set binds the form to a model and further streamlines the code.

The simple form helpers have the `_tag` suffix and include `form_tag`, `label_tag`, `text_field_tag`, and `submit_tag`.

```
<%= form_tag "http://google.com/", :method => "get" do %>
  <%= label_tag :q, "Find" %>
  <%= text_field_tag :q %>
  <%= submit_tag "Go" %>
<% end %>
```

The model-bound form helpers do not have the _tag suffix and can either specify the model as the first argument, or be prefixed with a block variable inside a form block and omit the model argument.

```
<%= form_for :hotel, @hotel, :url => hotels_path do |f| %>
  <%= f.text_field :name %>
  <%= f.text_area :description, :size => "40x4" %>
  <%= submit_tag "Create" %>
<% end %>
```

With RESTful resources, there is an even more succinct syntax for the form tag.

```
<%= form_for @hotel do |f| %>
```

Since most browsers only support the `GET` (used for index, show, new, and edit) and `POST` (used for create) HTTP verbs, Rails adds a hidden field called `_method` which tells Rails to treat a `POST` as a `PUT` when submitting a form to the update method and as a `DELETE` for the delete method.

Select Helpers

There are several ways to generate select and option tags. The simplest is to use `select_tag` and `options_for_select`. As with the generic form tags, no associated model is assumed.

```
<%= select_tag :size, options_for_select(["XS", "S", "M", "L", "XL"]) %>
```

In the case where the select is an attribute of a model, and the options are references to another model, collection_select is a succinct option.

```
collection_select(
  :hotel,
  :city_id,
  City.order(:name),
  :id,
  :name
)
```

Date Helpers

Date and time helpers introduce a slight mismatch between the model attribute and the form fields. A date would typically be represented by three fields, one each for year, month, and day. These separate parameters would then have to be converted into a single value in the controller before being assigned to the appropriate model attribute. Rails provides date helpers, in both generic and model-bound flavors, to help manage this complexity. For example:

```
<%= date_select :hotel, :established %>
```

will generate the three field names:

```
'hotel_established_1i', 'hotel_established_2i','hotel_established_2i'.
```

The Hotel object receiving these parameters will correctly recombine them into a single date attribute.

Note that there are two date helpers with confusingly similar names: `select_date`, which works with any Date object, and `date_select`, which is more commonly used, but can only be used with dates accessed through a model.

Walkthrough 7-1: The New Form

In this walkthrough you will create the `new` action and its corresponding form, as well as the `create` action.

Steps

1. Create `app/views/hotels/new.html.erb`.

```
<h1>New Hotel</h1>
<%= form_for @hotel do |hotel_form_builder| %>
  <div>
    <%= hotel_form_builder.label :name %>
    <%= hotel_form_builder.text_field :name %>
  </div>
  <div>
    <%= hotel_form_builder.label :description %>
    <%= hotel_form_builder.text_area :description, :size => '40x4' %>
  </div>
  <div>
    <%= hotel_form_builder.label :city_id, "City" %>
    <%= hotel_form_builder.text_field :city_id %>
  </div>
  <div>
      <%= hotel_form_builder.label :star_rating, "Rating" %>
      <%= hotel_form_builder.text_field :star_rating %>
  </div>
    <%=  hotel_form_builder.submit 'Save' %>
<% end %>
```

2. Add the `new` action to `app/controllers/hotels/controller.rb`.

```
def new
  @hotel = Hotel.new
end
```

3. Add the create action to `app/controllers/hotels/controller.rb` (insert after the `new` action)

   ```
   def create
     @hotel = Hotel.new(params[:hotel])
     if @hotel.save
        redirect_to hotels_path
     else
       render :text => @hotel.errors.full_messages
     end
   end
   ```

4. Try entering valid hotel data and saving. Try submitting an invalid hotel.

5. Improve the form by using the `collection_select` to display city options.

   We need to pass the `collection_select_helper` a collection of cities to display. Add the following line to the `new` action to retrieve the collection of cities and assign it to an instance variable so the View will have access to it:

   ```
   @cities = City.order(:name)
   ```

   Replace the city_id `text_field` with the following:

   ```
   <%= hotel_form_builder.collection_select(:city_id, @cities, :id, :name) %>
   ```

   The first parameter to `collection_select` represents the database column, the second is the collection to display, the third represents the attribute value for the selected record that should appear in the `params` hash, and the fourth represents the attribute value to display in the dropdown box for each record in the collection

   View the form in the browser. There's one small problem. It uses the first hotel as a default value. To fix this, specify a `:prompt` as follows:

   ```
   <%= hotel_form_builder.collection_select(:city_id,
       @cities, :id, :name, :prompt => "Select a city...") %>
   ```

   In the alternative you could specify `:include_blank => true` if you wanted the first line to be blank.

6. Now we will take care of the validation errors.

   First we'll address the navigation issues related to error handling and then we'll address the display issues.

   If we don't explicitly give Rails directions for where to go if the `save` fails, Rails will look for a file with a base name of "create" in the `app/views/hotels` directory by default. In order to display the new page replete with the users invalid data, we can use `render action: "new"`.

   The render method enables you to bypass the controller and display the template linked to the `new` action without invoking the action. If we were to use a `redirect` here,the `new` action would be invoked, which would wipe out any data the user entered in the hotel form data entry fields.

Replace the `render :text` call with `render action: "new"`.

If you try submitting the form with an invalid hotel, you will see an error in the console because we have not provided a collection of cities. The `@cities` variable we populated when the `new` action was invoked is no longer accessible. Add the following line to the branch of code that executes if the `save` fails as follows:

```
@cities = City.order(:name)
```

7. Next we'll add code to conditionally display any errors. Assign the error messages to a variable prior to the `render action: new` call as follows:

   ```
   @errors = @hotel.errors.full_messages.join("\n")
   render action: "new"
   ```

   Reference `@errors` in the view and conditionally display it as follows:

   ```
   <% if @errors %>
     <div class="error-details">@errors</div>
   <% end %>
   ```

   The view code references the `error` class to highlight the error message. Add the following to `app/assets/stylesheets/common.css.scss`

   ```
   .error-details {
     background-color: salmon;
   }
   ```

   Try entering an invalid hotel at this point. You should see error explanations, highlighted in red.

8. Now, we'll take care of highlighting the fields where data needs to be supplied or corrected. Enter an invalid hotel once again, and view the HTML source code via the browser. Search for name of one of the invalid fields. Notice that the field is wrapped in a `div` tagged with the class `field_with_errors`. When Rails detects errors on the object linked to the a form created with Rails helpers, it generates these elements for each field with a key in the `errors.messages` hash accessible following a failed `save`.

   Leverage the `field_with_errors` class to highlight the invalid values on the form, by specifying that `input`[text]`elements within`field*with*errors`elements should be highlighted to match the`div`displaying the validation error details. Where`error-details`is specified in`app/assets/stylesheets/common.css.scss‘ modify the file as follows:

   ```
   .error-details, .field_with_errors input[type="text"] {
     background-color: salmon;
   }
   ```

9. The `create` action should now be fully functional.

Walkthrough 7-2: Custom Form Builders

In this walkthrough you will create a custom form builder that enforces a theme for the application and lines up labels.

Steps

1. The form works, but looks ragged. The 'Description' label juts out passed the other labels and the text box appears to be unlabeled because the `Description` label is so far down.

   As a first step, we can create a CSS class for aligning the labels. Enter the following into `app/assets/stylesheets/common.css.scss`:

   ```
   .neat {
     float: left;
     text-align: right;
     width:75px;
     color: $base-color
   }
   ```

   Even though tagging labels with the `neat` class will not clutter up our forms as much as referencing each style individually, using the class attribute for each label on each form is not very DRY.

   We'll use a Rails custom form builder to make the forms more uniform throughout the application.

   A custom form builder typically extends the Rails default form builder to customize `text_field`, `label`, and other view helpers.

   Create a new file called `neat_form_builder.rb` in `app/helpers` and add the following `NeatFormBuilder` definition to it:

   ```
   class NeatFormBuilder < ActionView::Helpers::FormBuilder


     def label(method, text=nil, options = {})
       options[:class] = options.has_key?(:class) ?
                         "#{options[:class]} neat" : "neat"
       super(method, text, options)
     end


   end
   ```

   Specify that the custom form builder should be used in lieu of the default in the `form_for` call using the `:builder` option as follows:

   ```
   <%= form_for @hotel, :builder => NeatFormBuilder do |hotel_form_builder| %>
   ```

2. Take a look at the enhanced form.

Walkthrough 7-3: Forms with Nested Models (Think: Order/Items)

In this walkthrough you will get a feel for how to handle nested models in a Rails form.

Steps

1. Many hotels have more than one restaurant, so we're going to add a model to represent restaurants, and then modify the data entry form for Hotels so that the user can enter the hotel and any restaurants located in the hotel with a single "Submit."

   Use the following command to generate the restaurant model:

   ```
   rails g model restaurants name:string chef:string hotel_id:integer
   ```

   Run the migration.

   Add some validation to restaurant.rb. Use `validate_presence_of` to ensure that a restaurant cannot be saved without a :name.

   Specify the relationship between Hotels and Restaurants by adding the following line to `models/restaurant.rb`:

   ```
   belongs_to :hotel
   ```

   ... and the following line to `models/hotel.rb`:

   ```
   has_many :restaurants, :dependent => :destroy
   ```

   Also add the following `accepts_nested_attributes_for` call to `models/hotel.rb`:

   ```
   accepts_nested_attributes_for :restaurants,
     :allow_destroy => true,
     :reject_if => proc{|attrs| attrs[:chef].blank? && attrs[:name].blank? }
   ```

   The `accepts_nested_attributes_for` method enables you create a parent model and one or more children with a single call to `save` or `create`, provided the hash passed to `save` or `create` contains a key/value pair where the key consists of the child model's name pluralized, an underscore, and the word "attributes" (e.g. `:restaurants_attributes`) and the value is either an array of hashes containing attribute values for the child records or a hash of child record hashes keyed on identifiers for the child records -- e.g:

   ```
   {:name => "The Ritz", :star_rating => 5,
    :city_id => 2, :description => 'Like a home away from home.',
    :restaurants_attributes =>
      {"new_restaurant1" => {:name => "Ritz Cafe", "chef" => "John Doe"}}}
   ```

   If the `:allow_destroy` value is not set to `true`, as it is for the Hotels/Restaurants example , Rails will ignore programatic requests to delete child records with the Rails nested models API.

   Typically `:reject_if` is paired either a proc or method name that will return true if none of the significant properties of the child record are populated so that Rails can discard any child hash that is empty or contains little more than empty strings.

   If database identifiers are supplied in the child hash (ie `restaurants_attributes`), Rails will update the specified child record. If the child records are supplied as an array of hashes, the database identifier for an existing child record can be specified using an `:id` key in the child attributes hash. In the alternative, the child records can be supplied as a hash of hashes where the keys for existing child database records are their database ids (formatted as integers or strings) and the keys for new child records unique within the hash -- and the values are hashes of individual child record attributes.

You can see this behavior in the console.

Enter the following into the console to create a new hotel with a restaurant:

```
Hotel.create({:name => "The Ritz", :star_rating => 5,
              :city_id => 2, :description => 'Like a home away from home.',
              :restaurants_attributes =>
                {"new_restaurant" => {:name => "Ritz Cafe", "chef" => "John Doe"}}})
```

You should see the insert statement used to create the Hotel record and then the insert statement used to create the restaurant scroll by. Notice that Rails automatically populated the `hotel_id` column for the new Restaurant with the `hotels#id` value from the newly-created Hotel.

To add a new restaurant (Ritz Diner) and update the chef for the existing restaurant (to John Smith), you could call update_attributes on the newly-created Hotel. The command would look something like the following, where I've used `Hotel.find(:last)` to get the newly-created Hotel and Restaurant.find(:last).id to get the id for the newly-created Restaurant:

```
Hotel.find(:last).update_attributes({
    :name => "The Ritz",
    :star_rating => 5, :city_id => 2,
    :description => 'Like a home away from home.',
    :restaurants_attributes => {
       Restaurant.find(:last).id => {
        :name => "Ritz Cafe", "chef" => "John Smith"},
        "new_diner" => {:name => "Ritz Diner", "chef" => "John Doe"}}})
```

2. Next we'll look at how Rails handles nested models from the perspective of a view.

   Add the following panel for entering Restaurants prior to the "Save" button. Note that it contains a link for displaying a new set of data entry fields and a link for removing them.

```
<fieldset>
   <h3>Restaurants</h3>
   <%= hotel_form_builder.fields_for :restaurants do |restaurant_form_builder| %>
     <hr/>
       <div>
         <%= restaurant_form_builder.label :name %>
         <%= restaurant_form_builder.text_field :name %>
       </div>
       <div>
         <%= restaurant_form_builder.label :chef %>
         <%= restaurant_form_builder.text_field :chef %>
       </div>
     <hr/>
   <% end %>
</fieldset>
```

   When you view the data entry screen for Hotels, the Restaurants label shows up, but no fields. The `fields_for` helper generates HTML elements linked to the specified model but does not generate form tags. When `fields_for`is used in conjunction with a collection (in this case `restaurants`) its block argument is executed for each item in the collection. In order to provide data entry fields for Restaurants, we'll need to prime the Hotel instance we create in the `new` action with one or more restaurants.

Modify the `new` action as follows so that the restaurant entry section of the screen will be displayed:

```
def new
    @cities = City.order(:name)
    @hotel = Hotel.new
    3.times do
        @restaurants = @hotel.restaurants.build
    end
end
```

Now try hitting save with one valid restaurant and one valid one. Leave one of the sets of placeholder fields empty. Notice that the error detail text includes Restaurant details as well as Hotel details and that Rails correctly identifies the correct restaurant to highlight.

Rails will ignore the empty Restaurant element by virtue of the `reject_if` parameter proc we passed to `accepts_nested_attributes for`,

Correct the validation issues and save.

3. Rails does not provide much View support for nested models. As you have seen, there's no way for a user to enter more than 3 restaurant. A popular `nested_form` plugin, created by Ryan Bates, host of the RailsCast Screencasts, provides links for adding and removing data entry fields for new child records, and packages child record per the nest attributes API we just test-drove in the console.

   Install Jonhnny Weslley's fork of the `nested_form` plugin, because the master version maintained by Ryan Bates does not work with Rails 3.1.

   Add the following to Gemfile and run `bundle install`:

   ```
   gem "nested_form", :git => "https://github.com/jweslley/nested_form.git"
   ```

   Install the `nested_form` plugin by running the following command:

   ```
   rails g nested_form:install
   ```

   Add `//= require nested_form/jquery` to `app/assets/javascripts/application.js`

   Pass `nested_form` to `javascript_include_tag` in app/views/layouts/application.html.erb as follows:

   ```
   <%= javascript_include_tag "application", "nested_form" %>


   In 'app/views/hotels/_new.html.erb', replace 'form_for' with 'nested_form_for'
   ```

   The `nested_form` plugin mixes the 'link*to*remove' helper and the 'link*to*add' helper into the default Rails form builder. Since we're using a custom form builder, we'll need to add the following `include` statement directly under the `class` declaration in /helpers/neat*form*builder.rb' as follows:

```
class NeatFormBuilder < ActionView::Helpers::FormBuilder
include ::NestedForm::BuilderMixin
  ...
end
```

Add links for adding and removing Restaurants.

```
<fieldset>
    <h3>Restaurants</h3>
    <%= hotel_form_builder.fields_for :restaurants do |restaurant_form_builder| %>
      <hr/>
        <div>
          <%= restaurant_form_builder.label :name %>
          <%= restaurant_form_builder.text_field :name %>
        </div>
        <div>
          <%= restaurant_form_builder.label :chef %>
          <%= restaurant_form_builder.text_field :chef %>
              <%= restaurant_form_builder.link_to_remove "Delete" %>  <-- Add this line.
        </div>
      <hr/>
    <% end %>
      <%= hotel_form_builder.link_to_add "Add a Restaurant", :restaurants %>  <-- Add this line.
</fieldset>
```

4. Create a hotel through the GUI, and add a restaurant or two.

Walkthrough 7-3: Edit, Update and Delete

In this walkthrough you will complete the CRUD cycle.

Steps

1. In this step we'll handle `edit` and `update`. `Edit` and `update` need to do most of the things that `new` and `create` need to do. Fortunately we can easily reuse the form we created when we implemented `new`.

   As it turns out, we can reuse everything in `app/views/hotels/new.html.erb` except for the `h2` element containing a header for the page. Cut out everything below `<h2>New Hotel</h2>` and paste it into a new file -- `app/views/hotels/_form.html.erb` Now the `new` form and the `edit` form will be able to share this group of input fields and labels.

   Shareable View fragments are referred to as "partials" in Rails. The naming convention for partials is to prefix their names with an underscore.

   In order to display a partial, you pass it to the `render` method. To display the hotels partial in `index.html.erb`, insert the following text in the same spot in the file where the list of hotels initially lived: `render "hotels"`. Note that when you pass the name of a partial to the `render` method, you need to omit the underscore.

   Include the partial in `new.html.erb` with:

   ```
   <%= render "form" %>
   ```

   Make sure the application still works.

Create the `edit` view by creating `app/views/hotels/edit.html.erb` and enter the following 2 lines:

```
<h1>Edit a Hotel</h1>
<%= render "form" %>
```

Next we'll add the corresponding controller actions.

The `edit` action populate both the `@hotel` variable and the `@cities` collection needed by the select helper. But since the filter takes care of populating `@hotel`, `edit` can be a one-liner:

```
@cities = City.order(:name)
```

Finally, we'll add the `update` action. The `update` action is very similar to create, but instead of passing the `params` hash to `new`, we need to pass the params hash to `update_attributes` in order to persist the data and instead of rendering the `new` template when there are errors, we need to render the `edit` template.

Implement the `update` action as follows:

```
def update
  if @hotel.update_attributes(params[:hotel])
  redirect_to hotels_path
  else
  @errors = @hotel.errors.full_messages.join("\n")
  @cities = City.order(:name)
  render :action => "edit"
  end
end
```

Try to update hotel data. To update a Hotel with a `hotels#id` of 1, you would visit: `http://localhost:3000/hotels/1/edit`

2. To complete the cycle, add the `delete` action to `hotels_controller.rb`

```
def destroy
  @hotel.destroy
  redirect_to(hotels_url)
end
```

You can add support for multiple formats for one or more actions by refactoring the Hotels controller to use `respond_with` for those actions.

Lab 7:

Objectives

In this lab you will:

- Enhance existing forms
- Write the basic CRUD actions for Trips

Steps

1. Replace the star*rating text*field in with a radio button group containing 5 radio buttons. The radio button labels should be `"Perfect","Great","Good","Fair","Poor"`. If the user chooses `"Perfect"`, then 5 should be assigned to star*rating. If the user chooses* `"Great"`*, 4 should be assigned to star*rating, etc.

   Use the `radio_button` helper:
   `http://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html#method-i-radio_button`

2. Using the Hotel forms as a guide, add CRUD support for Trips. The data entry form used for Trips should include the name, date and hotel.

   Use the `date_select` helper:
   `http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html#method-i-date_select`

   Use the `collection_select` helper we used to display Cities to display a list of all the Hotels in the system. The documentation is here:
   `http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html#method-i-collection_select`

3. Add a trip by visiting `http://localhost:3000/trips/new`

Unit 8: Test-Driven Development

Unit Objectives

After completing this unit you should be able to:

- Explain the benefits of Test-Driven Development
- Understand the different test types used in a Rails application
- Write Unit tests using RSpec
- Use factories to facilitate object creation
- Write Integration tests using Rspec and Capybara

Unit Topics

- Test-Driven Development
- Standard Rails tests with Test::Unit
- RSpec
- Capybara
- Test Doubles, including mocks and stubs
- Factories

Test-Driven Development

TDD

Test-Driven Development, or TDD, popularized in the early 2000s by Kent Beck, is a process that inverts the traditional system of first writing code then testing it afterwards. By requiring the developer to describe the desired functionality first, TDD encourages loosely coupled, modular code. Having comprehensive test coverage, while not a replacement for QA, also facilitates refactoring by giving developers the freedom to experiment, knowing that if something breaks, they will find out immediately.

TDD is a set of practices, not a dogma, and 100% code coverage is a goal to strive for, not a mandate. Despite TDD's many benefits, even disciplined developers sometimes resist TDD because it does appear to require additional effort, particularly when first learning the process. In the long run, TDD invariably saves effort by reducing regression bugs and avoiding rework, but it does require an investment up front.

Note: Behavior Driven Development (BDD) is a popular extension of TDD. In this course, we will simply use the more generic term TDD to refer to both.

TDD Process

The TDD process consists of:

1. writing a failing test
2. writing the code to pass that test
3. refactoring (improving the code without changing the functionality) the code as needed.

The entire cycle known as "red/green/refactor."

First, identify the desired functionality and write a test that describes that functionality. This is sometimes expressed as testing the code "you wish you had." The test should fail, ensuring that the new test does in fact require new code. The reason for the test failure should also be consistent with the desired functionality.

Next, write the least amount of code required to make the test pass. The code does not have to be elegant or perfect. At each stage the entire set of tests should be run, ensuring that the new code does not interfere with any previously functioning code. Finally, the new code can be refactored as necessary.

Note that TDD does not, of course, guarantee good code or good software. Tests and code can be written poorly while following the steps outlined above. TDD is simply a process, like any other, which has both a spirit and a letter.

Rails Test Taxonomy

There are several categories of tests whose names are, confusingly, used differently in different contexts. Limiting ourselves to the world of Rails, tests are categorized as follows:

- Unit Tests, which test models in isolation.
- Functional Tests, which simulate HTTP requests (without going through the full Rails stack) to test controllers (and optionally views).
- Integration Tests, which do not bypass the Rails startup process and mimic the entire request cycle, making them slower than unit and functional tests, but necessary to test the full application stack.

In this class, we'll be doing unit tests and integration tests.

TDD Frameworks

Ruby and Rails have done a lot to promote TDD as a practice. The Test::Unit framework is built in to Ruby, and Rails is configured by default with Test::Unit support. Other popular test frameworks include MiniTest (which comes with Ruby 1.9 but is available for 1.8) and RSpec.

In this course, we will be using RSpec (a BDD framework), in part because its more natural semantics are more intuitive to grasp for newcomers to TDD. RSpec also offers other desirable features, such as customizable formatters, nested example groups and better component isolation. As an additional benefit, if you decide to use the Cucumber framework (a higher level BDD framework), you will find that Cucumber works seamlessly on top of RSpec.

Ultimately the choice of test framework is up to the individual developer and his or her team. Whichever framework helps you best do TDD is the best framework for you.

MiniTest and Test::Unit

Rails provides a testing framework known as Test::Unit (see `https://github.com/test-unit`). This framework provides a rake task, `rake test` with a number of targets, namely:

- `rake test:units' - run all unit tests, which are tests that live in the project's`test/unit' directory.
- `rake test:functionals` - run all functional tests, which live in 'test/functional'. These tests go against controllers and are able to run the stack without worrying about the web front-end. You submit controller requests and observe returned models in these tests, exercising the back-end system.
- `rake test:integration` - run all integration tests in 'test/integration'. These tests generally run the entire web stack and can involve web front-end testing tools like WebRat or Capybara.
- `rake test:performance` - runs performance tests. See the `test/performance` directory for details.

The slides for this chapter cover the standard Test::Unit test cases. However, the labs work with a more advanced testing and spec writing framework, RSpec.

RSpec Elements

RSpec is a DSL for creating executable examples of how code is expected to behave. It organizes your tests into groups, using the words "describe" and "it" so we can express our testing concepts conversationally.

A typical RSpec statement consists of an assertion and a matcher. An assertion is a way of stating that your code should or should_not have a specific behavior. A matcher is a special object that has methods for determining if your object matches an actual expected result, and a method for providing a failure message if it does not match.

Suppose we want to have a pillow object with a softness method. We might write the following test:

```
describe Pillow do
  before(:each) do
    @pillow = Pillow.new
  end

  it "should be soft" do
    @pillow.should be_soft
  end
end
```

The `describe` block wraps our tests for the `Pillow` object, and the `before` block contains setup code that is run before each test, in this case instantiating a new `Pillow` object and assigning it to the instance variable `@pillow`.

In addition to `before`, you can define `after` block that will run after the test runs. If you have expensive operations that you wish to run for all tests, you can use `before(:all)` and `after(:all)`.

Typically, an assertion is `should` or `should_not`, in this case `should`. There are many built-in matchers in RSpec including `==, exist, have, include, respond_to`, as well as the chameleon-like `be`, which dynamically matches any predicate in the object.

The minimal code required to make the test pass would simply be the following:

```
class Pillow
  def soft?
    true
  end
end
```

More on Predicates

In Ruby, a predicate is a method that ends with a "?" and returns a boolean value of true or false. You commonly see methods such as `empty?`, `nil?`, and `instance_of?` when dealing with Ruby objects.

To make tests match natural language more readily, RSpec will dynamically create matchers for you on the fly for any arbitrary predicate. That is where the aforementioned `be` comes into play.

```
# Ruby predicate
"Chunky Bacon".empty?  # Returns false

# RSpec assertion/matcher
"Chunky Bacon".should be_empty  # Fails
"Chunky Bacon".should_not be_empty # Passes
```

Additionally, you can use `be_a_` or `be_an_` instead of `be_` if it helps make your test read more naturally.

```
"A string".should be_an_instance_of(String)
```

The above will call `"A string".instance_of?(String)`.

Walkthrough 4-1: Working with RSpec

In this walkthrough, you will add a new spec for a simple model and practice using TDD, following the red/green/refactor principles.

Steps

1. Write a failing test. Create `spec/models/restaurant_spec.rb` and write a test to verify the existence of the Restaurant model.

   ```
   require 'spec_helper'


   describe Restaurant do
     it "should exist" do
        @restaurant = Restaurant.new
        @restaurant.should be_valid
     end
   end
   ```

2. Run the test. You should get an error because the Restaurant class does not exist at all yet.

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

   ... (should see an error) ...
   ```

3. Create the class to make the test pass. We'll use the generator to create the model file and the migration in one step. When asked if you want to overwrite `spec/models/restaurant_spec.rb`, type "n".

   ```
   $ rails g model Restaurant name:string description:text
   ```

   Run the test again. You would expect it to pass, but it does not. There is no error this time, but the test fails, complaining that the hotels table does not exist. If you run the migration to create the table, you will see the same error. The reason for this is that we have been running our migrations in the development environment, but not the test environment.

   RSpec is configured to run in the test environment. So run your migrations, this time specifying the test environment as well as development:

```
$ rake db:migrate

$ RAILS_ENV=test rake db:migrate
```

4. Run the test again. You should see a green dot, indicating one successful test, followed by a summary of the results.

5. Add some validations to the Restaurant model by first writing a failing test. Edit `specs/model/restaurant_spec.rb` by replacing the original test with the following:

```
require 'spec_helper'


describe Restaurant do
  it "should be valid with valid attributes" do
    @restaurant = Restaurant.new(name: "Bob's BBQ")
    @restaurant.should be_valid
  end


  it "should not be valid without a name" do
    @restaurant = Restaurant.new
    @restaurant.should_not be_valid
  end
end
```

6. Run the test. It should fail.

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

7. Add the validation to `app/models/restaurant.rb`

```
class Restaurant < ActiveRecord::Base
  validates :name, :presence => true
end
```

8. Add another test for the second validation.

```
it "should not be valid with a name shorter than 2 characters" do
  @restaurant = Restaurant.new(:name => "A")
  @restaurant.should_not be_valid
end
```

9. Run the test. It should fail.

10. Add the second validation:

```
class Restaurant < ActiveRecord::Base
  validates :name, :presence => true
  validates_length_of :name, :minimum => 2
end
```

11. The test should now pass.

12. Create the City model from the walkthrough in Unit 4 if you have not already done so.

13. Write a failing test for the association between Restaurant and City in `spec/models/restaurant_spec.rb`. Each restaurant is located in a single city.

```
it "should have a city" do
  @restaurant = Restaurant.new()
  @restaurant.should respond_to(:city)
end
```

14. Run the spec to verify it fails.

15. Add the association to `app/models/restaurant.rb`.

```
class Restaurant < ActiveRecord::Base
  validates :name, :presence => true
  validates_length_of :name, :minimum => 2
  belongs_to :city
end
```

16. The test should now pass.

17. Write a failing test for the association between City and Restaurant in `spec/models/city_spec.rb`. Each city can have multiple restaurants.

```
require 'spec_helper'


describe City do
  it "should have multiple restaurants" do
    @city = City.new
    @city.should respond_to(:restaurants)
  end
end
```

18. Run the test to verify it fails.

19. Add the association to `app/models/city.rb`.

```
class City < ActiveRecord::Base
  has_many :hotels
end
```

20. Write a migration to add a foreign key column to the restaurants table.

```
$ rails generate migration AddCityReferenceToRestaurants
```

21. Add the new column to the up and down methods in the generated migration file.

```
def change
  add_column :restaurants, :city_id, :integer
end
```

22. Run the migration and tests to make sure it all passes.

Testing Controllers

In this class, we won't test our controllers in isolation as we did with our models. Instead, we'll test the whole stack — we'll try to interact with the application as a user would, which may involve testing model and view code as well as controller code.

We are making this choice because integration testing (testing the full stack) often shows issues that functional testing (controller-focused testing) does not. Furthermore, it's a good programming practice to put the vast majority of logic in model code ("fat models").

This is not to say that there is nott value in testing controllers in isolation, but the combination of unit tests for models and integration testing is a relatively easy way to get started which will provide a good degree of test coverage. That said, there are some techniques for testing code in isolation that bear mentioning.

Capybara

Testing controller actions in integration tests requires testing the flow of the application as if your test runner were a user sitting at a web browser. The Capybara gem handles this by providing a DSL for visiting web pages, interacting with forms, and viewing the resulting page. Capybara relies heavily on XPath, which is used to navigate through elements and attributes in an XML document with relative ease.

To visit a page using Capybara, in your test spec you can use the `visit` method:

```
begin do
  visit('/hotels')
end
```

You can also use helper methods provided by Rails to keep from hardcoding URLs.

```
begin do
  visit(hotels_path)
end
```

This will perform a GET request to `/hotels` and return a `page` object. You can also navigate pages, click links, and fill out forms using simple methods. Most of these methods take a locator as the first parameter used to find the correct field, form object, or link. A locator can either be the object's ID, name, or valid label text.

- `click_link('Link Text')`
- `click_button('Save')`
- `click_on('Link Text')`
- `fill_in('First Name', :with => 'Tim')`
- `fill_in('Password', :with => '12345')`
- `fill_in('Description', :with => 'I have the same combination on my luggage!')`
- `choose('A Radio Button')`
- `check('A Checkbox')`
- `uncheck('A Checkbox')`
- `attach_file('Image', '/path/to/image.jpg')`

- `select('Option', :from => 'Select Box')`

Once you have navigated the page successfully, you can use the page object to check the results of your navigation.

- `page.has_xpath?('//div/span')`
- `page.has_css?('div span.foo')`
- `page.has_content?('foo')`
- `page.html # raw HTML output`
- `page.all('a') # Returns an array of all a elements`

Capybara works well with RSpec. You can use the page object as a matcher for the normal assertions.

- `page.should have_xpath('//div/span')`
- `page.should have_css('div span.foo')`
- `page.should have_content('foo')`

Test Doubles: Mocks, Stubs, Fakes, and their friends

Testing controller actions requires interaction with models. We could simply instantiate real models in our controller tests, but it is preferable to isolate components when not doing integration testing.

Creating models for controller tests not only creates a dependence on model code (if the test fails how do we know if it's the fault of the model or the controller), it also slows down the tests because instantiating, saving, and loading the models is expensive, resource-wise.

Another issue we may face is that our code may need to access an external resource (e.g. API) that we don't want to access when testing. Using test doubles (things that stand in for the actual objects, we can speed things up, isolate tests, and often have shorter tests.

Mocks are imitation objects with defined behavior and expectations. Instead of instantiating the actual model, we create a class and define its output in response to specific input. In the case of mocks, not only is the behavior defined, there is an expectation that the mock will receive the input specified.

Stubs provide canned answers to calls similar to mocks. The main difference between mocks and stubs and is that stubs have no expectations. Mocks will complain if expectations are not met. Stubs never complain. A test will fail if a mock's expectations are not met. Stubs will never cause a test to fail.

Fakes are simulations of objects that do not necessarily have any expectations associated with them. A common use of fakes is to simulate an external server. If your application makes use of a Web API, you can use a fake to help you simulate the actual service (sometimes with recorded information). Some to stand in for the actual service.

There are many test double libraries that exist for Ruby. Rspec provides its own, but makes it easy to substitute others such as RR (Double Ruby), Mocha, and Flex Mock. Some libraries aimed at simulating http requests include FakeWeb and WebMock. Another project called VCR records HTTP sessions.

Walkthrough 8-2: Integration Testing and Creating Controller Actions

In this walkthrough you will create index and show controller actions for the hotels_controller and update the index and show views to use dynamic data.

Steps

1. Let’s make some changes to our Gemfile:

```
source 'http://rubygems.org'


gem 'rails', '3.1.0.rc5'
gem 'sqlite3'


# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', "~> 3.1.0.rc"
  gem 'coffee-rails', "~> 3.1.0.rc"
  gem 'uglifier'
end


gem 'jquery-rails'


group :test, :development do
  # To use debugger
  # gem 'ruby-debug19', :require => 'ruby-debug'


  gem "factory_girl_rails", ">= 1.1.0"
  gem "factory_girl_generator", ">= 0.0.3"
  gem "rspec-rails", ">= 2.6.1"
  gem 'RedCloth', '~> 4.2.7'
  gem "capybara"
end
```

   The above adds the capybara gem and also reorganizes our Gemfile a little bit by putting our development and testing gems in their own block.

2. Install the new gems:

```
$ bundle install
```

3. Write failing tests for `index` in `spec/integration/hotels_spec.rb`:

```
require 'spec_helper'


describe "hotels" do
  describe "listing hotels" do
    before do
     @hotel1 = Hotel.create(name: "The Lakes")
```

```
      @hotel2 = Hotel.create(name: "The St. Clair")
      visit "/hotels"
    end
    it "should list as many hotels as we have" do
      page.all('ul li').should have(2).hotels
    end
    it "should list all hotel names (xpath)" do
      page.should have_xpath("//li[.='#{@hotel1.name}']")
      page.should have_xpath("//li[.='#{@hotel2.name}']")
    end
    it "should list all hotel names (css)" do
      hotel_names = page.all('ul li').map(&:text)
      hotel_names.should include(@hotel1.name)
      hotel_names.should include(@hotel2.name)
    end
  end
end
```

Note that the last two examples do the same thing. The difference is that one finds hotels in the HTML using CSS and the other uses XPath.

4. Run the spec and watch it pass.

   ```
   $ rspec spec/integration/hotels_spec.rb
   ```

5. Now let's add a show view to display an individual hotel. First, we start by adding a new example:

```
require 'spec_helper'


describe "hotels" do
  describe "listing hotels" do
    before do
     @hotel1 = Hotel.create(:name => "The Lakes")
     @hotel2 = Hotel.create(:name => "The St. Clair")
     visit "/hotels"
    end
    it "should list as many hotels as we have" do
      page.all('ul li').should have(2).hotels
    end
    it "should list all hotel names (xpath)" do
      page.should have_xpath("//li[.='#{@hotel1.name}']")
      page.should have_xpath("//li[.='#{@hotel2.name}']")
    end
    it "should list all hotel names (css)" do
      hotel_names = page.all('ul li').map(&:text)
      hotel_names.should include(@hotel1.name)
      hotel_names.should include(@hotel2.name)
    end
    describe "when clicking on a link" do
      before do
        click_link(@hotel1.name)
      end
      it "should show detailed information for our hotel" do
        page.should have_xpath("//h1[.='#{@hotel1.name}']")
      end
      end
  end
end
```

6. Run the test and it should pass.

Factories

Our integration test only has two hotels in it. For something so simple, creating the hotels from scratch isn't so bad. But as our models get more complex, so will our code.

Factories address this problem by making it easy to build or create new models. In this class, we use factory_girl, but there are several libraries that do something similar. These include Fixjour, Machinist, and FixtureReplacement.

Let's add a few requirements to our models. We'll add validations to make sure that each Hotels will only be valid if they include a City. Likewise, each Trip must have an associated Hotel.

Walkthrough 6-4: Using Factories

Steps

1. Write some failing tests. Since this is a change to our models, we'll start with failing model specs. Modify `spec/models/hotel_spec.rb`.

```
require 'spec_helper'


describe Hotel do
  before do
    @detroit = City.create(:name => "Detroit")
  end
  describe "validations" do
    it "should be valid with valid attributes" do
      @hotel = Hotel.new(:name => "The St. Clair", :city => @detroit)
      @hotel.should be_valid
    end
    describe "invalid cases" do
      before do
        @hotel = Hotel.new
        @hotel.should_not be_valid
      end
      it "should not be valid" do
        lambda {@hotel.save!}.should raise_error(ActiveRecord::RecordInvalid)
      end
      it "should not be valid without a name" do
        @hotel.errors.full_messages.should include("Name can't be blank")
      end
      it "should not be valid with a name shorter than 2 characters" do
        @hotel.errors.full_messages.should include("Name is too short (minimum is 2 characters)")
      end
      it "should have a city" do
        @hotel.errors.full_messages.should include("City can't be blank")
      end
    end
    it "should have many trips" do
      Hotel.new.should respond_to(:trips)
    end
  end
  describe "#yet_to_visit" do
    before do
      @visited_hotel = Hotel.create(:name => "Been There Motel", :visited => true, :city => @detroit)
      @unvisited_hotel = Hotel.create(:name => "Let's Go Lodge", :visited => false, :city => @detroit)
    end
    it "should return unvisited hotels" do
      Hotel.yet_to_visit.should eq([@unvisited_hotel])
    end
  end
end
```

2. Run the spec and watch it fail.

3. Modify the Hotel model to get it to pass:

   ```
   validates_presence_of :city
   ```

4. Now let's run all the specs:

   ```
   $ rspec spec
   ```

5. Our integration spec is now failing. If we had a whole lot more integration specs we might have a lot more errors.

   Let's address this by adding factories that will create valid models. If you have a `spec/factories/` or `test/factories/` directory, delete them. We're going to put all of our factories in one place: `spec/factories.rb`. Create the file with this content:

   ```
   Factory.define :city do |f|
     f.sequence(:name) { |n| "city #{n}" }
   end
   Factory.define :hotel do |f|
     f.sequence(:name) { |n| "hotel #{n}" }
     f.association :city, :factory => :city
   end
   ```

6. Now let's modify our `spec/integration/hotels_spec.rb` to use the new factories (just the before block needs to be changed):

   ```
   require 'spec_helper'


   describe "hotels" do
     describe "listing hotels" do
       before do
         @hotel1 = Factory.create(:hotel, name: "The Lakes")
         @hotel2 = Factory.create(:hotel, name: "St. Clair Hotel")
         visit "/hotels"
       end
       it "should list as many hotels as we have" do
         page.all('ul li').should have(2).hotels
       end
       ...
   ```

7. Run all the specs again and there should be no failures!

Unit Review

- Test-Driven Development
- RSpec
- Capybara
- Test Doubles, including mocks and stubs
- Factories

Lab 8: Test-Driven Development

Objectives

- Write a passing integration test using Rspec and Capybara

Steps

1. Write an integration spec for the Trips controller.
2. Modify the spec for your Trip model. Modify `spec/models/trip_spec.rb`:

```
require 'spec_helper'


describe Trip do
  describe "validation" do
    before do
      @detroit = City.create!(name: "Detroit")
      @st_clair_hotel = Hotel.create!(name: "St. Clair Hotel", city: @detroit)
    end
    it "should be valid with valid attributes" do
      @trip = Trip.new(name: "Visit Shanghai", hotel: @st_clair_hotel)
      @trip.should be_valid
    end
    describe "when invalid" do
      before do
        @trip = Trip.new
        @trip.should_not be_valid
      end
      it "should raise an error when receiving #save!" do
        lambda {@trip.save!}.should raise_error(ActiveRecord::RecordInvalid)
      end
      it "should not be valid without a name" do
        @trip.errors.full_messages.should include("Name can't be blank")
      end
      it "should not be valid with a name shorter than 2 characters" do
        @trip.errors.full_messages.should include("Name is too short (minimum is 2 characters)")
      end
      it "should have a hotel" do
        @trip.errors.full_messages.should include("Hotel can't be blank")
      end
    end
  end
end
```

3. Now run the spec and watch it fail.
4. Modify the Trip model to get the the spec to pass.
5. Run the integration spec to watch it fail.
6. Add a factory for the Trip model.
7. Modify the integration test for the Trips controller to use the factory.
8. Run the integration test again and watch it pass!

References from Unit 8

1. Paraphrased from James Grenning, http://www.renaissancesoftware.net/blog/

2. Including classes which are not descendants of ActiveRecord

3. You can also use the rake task db:test:prepare to copy the structure of the development database to the test database.

4. While not covered heavily in this class, w3schools has a great XPath tutorial:&#8232; http://www.w3schools.com/xpath/default.asp

5. https://github.com/thoughtbot/factory_girl

6. https://github.com/nakajima/fixjour

7. https://github.com/notahat/machinist (link is to Machinist 2, which is still in beta; there's a link on the page to Machinist 1)

8. https://github.com/smtlaissezfaire/fixturereplacement

Unit 9: Cloud Deployment

Unit Objectives

After completing this unit you should be able to:

- Set up a Github account
- Understand public key encryption
- Fork a Github repository
- Make modifications to source code
- Submit source code changes
- Deploy to the Cloud

Unit Topics

- OpenSSH public key encryption
- Forking Repositories
- Modifying Source
- Cloud Deployment

Github and Public Key Encryption

We utilize Github for this unit due to the many advantages of social coding it provides. Developers normally create a local git repository for the project, make and commit changes as they go, and push the code up to Github. From there, the code can be pulled from Github to the cloud as part of the deployment process.

You will need a Github account, but never fear as there is a free option available. The main limitation of the free plan is that you cannot have any private repositories, which is useful to have if your project is not open sourced.

To get started, go to http://github.com and register for a free account. Once you have an account, you're almost ready to go. The final step is ensuring that there is a secure connection between your machine and Github, and that requires the use of public key encryption.

NOTE: If you are a Windows user, the recommended approach for working with Github from the command line is to use git-bash, which is a Bash shell and working Unix utilities for Windows. Git is installed for you when you use the RailsInstaller package from http://www.railsinstaller.org.

You can also use PuTTY along with PuTTYgen and Pageant to manage the keys, but it will not be a consistent experience for this class.

Setting up an SSH public key (Mac/Linux/git-bash)

Public key encryption consists of two parts: the private key, which you keep to yourself, and the public key, which is given out to verify your identity. Before we continue, let's make sure you don't already have a public key:

```
$ ls ~/.ssh
```

```
authorized_keys
config
id_rsa
id_rsa.pub
known_hosts
```

In this example, I already have a private and public key, as denoted by the files `id_rsa` and `id_rsa.pub`. The `id_rsa.pub` file is the public portion of the key, while the other is the private portion and should not be shared with others.

If you do not have a key, or if you wish to rename the existing files and re-generate a new one, you can do so from the command line `ssh-keygen` utility:

```
$ ssh-keygen -t rsa -C "tgourley@engineyard.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/tim/.ssh/id_rsa): <press enter>
Enter passphrase (empty for no passphrase): <press enter>
Enter same passphrase again: <press enter>
Your identification has been saved in (/Users/tim/.ssh/id_rsa.
Your public key has been saved in (/Users/tim/.ssh/id_rsa.pub.
The key fingerprint is:
67:e0:9e:ef:af:08:a2:d3:88:be:a6:e1:9e:ec:64:8a tgourley@engineyard.com
The key's randomart image is:
+--[ RSA 2048]----+
|        .        |
|       . .       |
|        S o      |
|       . +       |
|.+ o. . o        |
|Oo+... . o       |
|EOo.    ..+o.    |
+-----------------+
```

Now you can open the file `id_rsa.pub` in a text editor, copy the contents, and add them to your Github account under “Account Settings” -> “SSH Public Keys” -> “Add another public key”

To test your key, ensuring everything will work smoothly when working with Github, you can attempt to ssh into github.

```
$ ssh git@github.com
```

```
The authenticity of host 'github.com (207.97.227.239)' can't be established.
RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? <strong>yes</strong>

Hi username! You've successfully authenticated, but GitHub does not provide shell access.
Connection to github.com closed.
```

Forking a Repository

One of the advantages of open source software is that developers can fork other people’s projects. A great example is the Apache Project, which forked the original NCSA web server, and took it in a new direction.

Forking

During this Unit, you will work with a new application called ey-roulette, which you will fork. Forking copies the repository from a different Github account to yours.

Pull Requests

Github saves the relationship between your forked repository, and the original, in case you ask that author to pull your code modifications into the forked repository. This is called a Pull Request.

Branching

Branching is a way to try new things without changing the master version of your code. After experimenting with a branch, you can either throw it away, or merge it with the master. When you fork a repository, you will be on the master branch of your fork. It is a good idea to branch your changes after you fork, especially if you intend on making a Pull Request. The name of your branch helps people understand the nature of your modifications.

Walkthrough 9-1: Forking a Repository

In this walkthrough, you will either register, or log in to Github, find the ey-roulette application, and fork it. Then you will copy the forked version down to your local disk and make sure that it works.

Steps

1. Visit Github at http://github.com
2. If you have an account, login. If not, register for a free account, login, and set up your keys as described in this unit.
3. Load the ey-roulette Github page: http://github.com/mreider/ey-roulette
4. Locate and click the **Fork** button
5. After a bit, your forked copy of ey-roulette appears under your Github account
6. Copy the **SSH URL** to your clipboard. This URL has the format of:

   ```
   git@github.com:student/ey-roulette.git
   ```

7. Open a Terminal
8. Clone your forked repository

   ```
   git clone git@github.com:student/ey-roulette.git
   ```

Modifying Source

Git in general, and Github in particular, are decentralized version control systems. The philosophy behind decentralized version control, and its role in helping the open source movement, is four-fold.

1. Source code is useless if it is stuck on people's hard drives
2. It should be easy for developers to find one another's work
3. Changes to source code should be easy to understand
4. Integrating those changes should also be easy

The first two points are obvious to anyone spending time using Github. The last two points are understood by those who choose to fork, modify, and submit pull requests.

Submitting changes to another person's public repository is both welcome and encouraged on Github. You can start out slowly, and modify things like README documentation, or you can write tests, reveal problems, and fix them. The point is to get involved, and try not to be shy. People appreciate help!

Walkthrough 9-2: Modifying Source

In this walkthrough you will test the ey-roulette application, make some changes, and commit those changes to your repository.

Steps

1. Make sure you are in the ey-roulette directory install gems using bundler

```
bundle install
```

2. Run the database migration

```
rake db:migrate
```

3. Start the application

```
rails s
```

4. View the application in a browser

```
http://localhost:3000
```

5. Open `public/stylesheets/styles.css` in a text editor
6. Change the font body color from #ffffff to #ffff00
7. In a terminal add the changes to your local repository

```
git add public/stylesheets/styles.css
```

8. Commit your changes

```
git commit -m 'changed fonts from white to yellow'
```

9. Push to your git repository

```
git push
```

Walkthrough 9-3: Branching

In this walkthrough you will branch the ey-roulette application, commit it to your repository and push. Next you will make a pull request of the original author.

Steps

1. Create a new branch of your local repository with an intuitive name.

```
git branch fonts
```

2. Switch to the fonts branch

```
git checkout fonts
```

3. Open `public/stylesheets/styles.css` in a text editor
4. Change the `a` (link) font color from `#ffffff` to `#ffff00`
5. In a terminal commit the changes to your local repository

```
git commit -a -m 'changed more fonts'
```

6. Switch back to the master.

   ```
   git checkout master
   ```

7. Look back at the `styles.css` page. Your changes are not on the master.

8. Push the new branch to Github

   ```
   git push origin fonts
   ```

Walkthrough 9-4: Making a Pull Request

In this walkthrough you will branch the ey-roulette application, commit it to your repository and push. Next you will make a pull request of the original author.

Steps

1. Go to your Github repository for ey-roulette.

   ```
   http://github.com/student/ey-roulette
   ```

2. Switch to fontcolors by selecting **switch branches** under **Source**

3. Click the **Pull Request** Button

4. Write a note to the author (he works for Engine Yard University)

5. Click **Send Pull Request**

6. If this were a real Pull Request, the author would correspond with you look at the code you wrote, and then merge it into his repository.

Cloud Deployment

The most common definition of Cloud Computing is the ability to use processing and storage without having to know where your computers are located, or how they are configured. But Cloud Computing is a multilayered technology, which has different meaning depending on the layer you are describing.

IaaS

At the heart of Cloud Computing are the Infrastructure as a Service (IaaS) providers such as Amazon EC2. These vendors operate massive data centers, and provide virtualized machines that are billed based on resources consumed, much like a utility company.

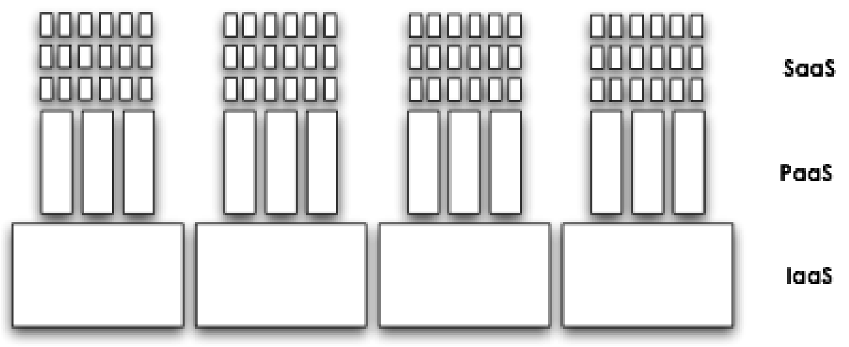
PaaS

On top of the IaaS layer are the Platform as a Service (PaaS) vendors, who provide a specific solution stack on which people can deploy their applications. This is the layer you will use in this unit.

Deploying an application to the cloud is a multi step process. First, you sign up for an account using a coupon code you will receive from your instructor. Once you are signed up there is a Quick Start, which we will walk through in a moment.

SaaS

On top of the PaaS providers are the Software as a Service (SaaS) vendors. This is the layer that people think of as "The Cloud," because it is expressed through a user interface, such as a photo sharing app, or social network.



Walkthrough 9-5: Cloud Deployment

In this walkthrough you will test the ey-roulette application, make some changes, and commit those changes to your repository.

Steps

Beginning the Quick Start

1. If you are already signed into the Engine Yard Cloud, sign out by visiting: `https://login.engineyard.com/` and clicking the Logout link. Alternatively you can clear your private browsing history, use Chrome's "Incognito mode", or a different browser to ensure you do not have the Engine Yard cookies lingering for your existing account.

2. Sign up for the Engine Yard Cloud at `http://cloud.engineyard.com`

   You will not need to enter any credit card information and will be eligible to use the Free Trial account with up to 500 hours of compute time.

3. You should be viewing step 1 of the Quick Start. If not click **Quick Start**

4. Add your forked git repository to the **Application Information** page

   ```
   git@github.com:student/ey-roulette.git
   ```

5. Choose Rails 3 as the application type
6. Accept the defaults for other options and click **Next**
7. Select **Passenger** as the Web Server Stack
8. On the next page is your Git Deploy Key. Copy the key to your clipboard
9. Return to your forked repository on Github

   `http://github.com/student/ey-roulette`
10. Click on the **Admin** button
11. Under **Repository Options**, click on the **Deploy Keys** link
12. Click **Add** a deploy Key
13. Name the key `my_key`
14. Paste the Github deploy key you copied from Engine Yard Quick Start
15. return to the Quick Start and indicate that your key is in place
16. Click **Next**
17. The next page is to configure Gems. Since we use Bundler, click Next

Adding a Public key

`If you did not create a public key earlier in this unit, do so now.`

1. Open `~/.ssh/id_rsa.pub` in a text editor
2. Select all of the text and copy it to your clipboard
3. Return to the Engine Yard Quick Start and name your key `cloud_key`
4. Paste your public key into the text area and click **Next**

Creating an Instance

1. The next page asks for **Configuration** options. Select **Custom**
2. Accept all defaults except Server Size, which should be Medium
3. Click **Boot This Configuration**
4. In roughly five minutes you should have an instance running

Using Ey Deploy

1. Open a terminal and install the Engine Yard Gem

   ```
   gem install engineyard
   ```

2. Make sure you are in the root directory of the ey-roulette application

3. Deploy your application

   ```
   ey deploy -e eyroulette_production
   ```

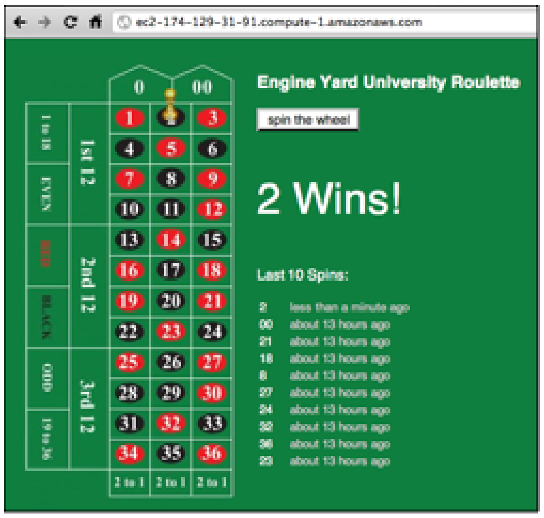
   Note: If you are a Windows user, you will need to use git-bash on Windows to perform this operation.

   Note: Your first use of `ey deploy` prompts for your email and password.

4. Return to your cloud dashboard: http://cloud.engineyard.com

5. Click the **HTTP** link to view your deployed application

6. You should see the roulette application in a browser:



Engine Yard University Roulette

2 Wins!

Last 10 Spins:

```
2   less than a minute ago
00  about 13 hours ago
21  about 13 hours ago
18  about 13 hours ago
8   about 13 hours ago
27  about 13 hours ago
24  about 13 hours ago
32  about 13 hours ago
36  about 13 hours ago
23  about 13 hours ago
```

Unit Review

- Public Key encryption
- Forking a Repository
- Modifying Code
- Deploying to the Cloud

References

1. Wikipedia has a decent article explaining public key encryption: http://en.wikipedia.org/wiki/Public-key_cryptography

Unit 10 (Optional): Configuration & Chores

Unit Objectives

After completing this unit you should be able to:

- Edit configurations for the three standard rails environments
- Write custom rake tasks
- Describe various code analysis tools
- Use metric_fu to analyze the application

Unit Topics

- Environments
- Rake Tasks
- Code Analysis

Environments

A standard Rails application has three environments: development, test, and production. As discussed when setting up the databases in Unit 3, each environment has its own separate database. All environments share the setup code in `boot.rb`, `environment.rb`, and `application.rb`, and each environment has its own configuration file in `config/environments`. If you replace core components of Rails, as we have done with the template engine and test framework, the generators are configured in `Application.rb`.

```
config.app_generators do |g|
  g.template_engine :haml
  g.test_framework :rspec, :fixture => true, :views =>; false
end
```

`Application.rb` also contains configuration information for default language and timezone if you support internationalization and localization in your application. There is also an option to strip specific parameters from logs, such as passwords and other sensitive information.

The individual environment files allow you to specify environment-specific configuration. Most notable is `config.cache_classes`, which reloads the application code on each request when set to `false`. In `development` this allows you to see changes to your code without having to restart the server. In `production` this is usually set to true to improve response time.

Rake Tasks

Rake, written by Jim Weirich, is a build tool similar to make. In Rails, rake is commonly used, as you have already seen, for common tasks like migrations and tests. `rake -T` provides a list of available rake tasks. You can write your own tasks and save them with the `.rake` suffix in the `lib/tasks/` directory.

A single `.rake` file can contain many tasks. Each task consists of a task block with an optional but recommended desc line (which is displayed as the description when you run `rake -T`).

```
desc "Prints a short message"
task :hello do
  puts "Hello, world."
end
```

Tasks can also invoke other tasks.

```
desc "Prints two short messages"
task :meet do
  Rake::Task['hello'].invoke
  puts "Nice to meet you."
end
```

Rake tasks can also be namespaced to avoid collisions with other task names or simply to group them logically (e.g. `rake db:migrate`, `rake:db:schema:dump`, etc.). The environment can also be loaded if needed before running a given rake task.

```
namespace :traveljournal do
  desc "Prints three messages"
  task :salutation => :environment do
    Rake::Task['hello'].invoke
    puts "Nice to meet you."
    puts "You are in the #{RAILS_ENV} environment."
  end
end
```

Walkthrough 10-1: Writing a Rake Task

In this walkthrough, you will write a simple rake task to drop and re-create your database from scratch. Use with caution, as it will destroy the existing contents of your database.

Steps

1. Create `traveljournal.rake` in `lib/tasks`.

   ```
   namespace :db do
     desc "Wipe database clean and start over"
     task :total_reset => :environment do
       Rake::Task['db:schema:dump'].invoke
       Rake::Task['db:drop'].invoke
       Rake::Task['db:create'].invoke
       Rake::Task['db:schema:load'].invoke
       Rake::Task['db:seed'].invoke
     end
   end
   ```

2. Run the task in the test environment

   ```
   RAILS_ENV=test rake db:total_reset
   ```

Code Analysis

There are many tools for analyzing code and identifying potential problems or areas that can be streamlined or optimized.

Code analysis tools can be a helpful a guideline, but are no substitute for your own judgement. Don't feel compelled to implement the tools' recommendations without understanding what and why. Take the time to investigate, but in the end, you may reasonably disagree with the output of a mechanical analysis. You may have a good reason for writing your code the way you have.

Metric_fu

Metric_fu is a gem that conveniently packages up six popular tools along with rake tasks for them.

Churn

Churn examines how frequently files are changing based on information from version control and flags files that are changing a lot as a potential source of complexity or trouble.

Flay

Flay identifies areas of code that are literally duplicates or structurally similar for possible refactoring and elimination of duplication.

Flog

Flog measures code complexity based on the numbers of assignments, branches, and calls.

Rcov

Rcov measures test coverage (C0) and identifies lines of code that it believes are not covered by unit or functional tests (it can be configured to factor in integration tests as well). Bear in mind that rcov is not yet 100% reliable on Ruby 1.9 and may report inaccurate results. If you are using Ruby 1.9 and want code coverage, check out the cover_me gem.

Reek

Reek detects common code "smells" such as duplication, nested iterators, poorly documented code, and code with low cohesion. Reek can be a little overzealous in identifying code that many would consider clean and clear.

Roodi

Roodi (Ruby Object Oriented Design Inferometer) catches common potential code issues such as assignment inside a conditional, cyclomatic complexity, overly long methods, and adherence to naming conventions.

Walkthrough 10-2: Using Metric_fu

In this walkthrough you will install the metric_fu gem, create a custom rake task to call it with the appropriate arguments for Rails 3, run it, and view the output.

Steps

Install metric_fu

1. Add a line in Gemfile to include the `metric_fu` gem.

   ```
   gem 'metric_fu'
   ```

2. Install the gem using bundler

   ```
   bundle install
   ```

3. Create `app/lib/tasks/metrics.rake`.

   ```
   namespace :tj do
     desc "Run metrics with proper rcov configuration"
     task :metrics do
       MetricFu::Configuration.run do |config|
         config.rcov[:rcov_opts] << "-Ispec"
   ```

```
     end
     Rake::Task['metrics:all'].invoke
    end


    desc  "Run specs with rcov"
    RSpec::Core::RakeTask.new("rcov") do |t|
      t.rcov = true
      t.rcov_opts = %w{--rails --include views -Ispec --exclude gems\/,spec\/,features\/,seeds\/}
    end
  end
```

Run and review metrics

1. Run metrics

   ```
   rake tj:metrics
   ```

2. Review metrics in browser

3. Refactor as appropriate.

Unit Review

- Environments
- Rake Tasks
- Code Analysis

The Rails Asset Pipeline

In the past, Rails has relied on serving up assets by using files inside the `public_html` directory. All of your images, Javascript files, stylesheets, and so on would go inside this directory, to be served statically by the application. If you ever wanted to minify, compress, or obfuscate the code in your Javascript or CSS files, you'd have to rely on external tools or third-party gems. In short, it was a lot of work.

Now Rails comes with an asset pipeline, which provides a framework to concatenate, minify, and compress your Javascript and CSS assets. It also allows you to write your assets in other languages such as CoffeeScript, SCSS or SASS, and even use template languages such as ERB.

Rails uses the Sprockets library to achieve this pipeline. Previously Sprockets was available by using the `sprockets-rails` gem, but now it is part of the core Rails setup and enabled by default. Developers can take advantage of these features without any additional configuration.

Using the pipeline, your assets will no longer be in `public_html`, but instead will reside in an `assets` directory in one of three locations:

- **`app/assets`** - Assets owned by the application, such as custom images, Javascript, or stylesheets. The bulk of your assets will go here.
- **`lib/assets`** - Any assets for custom libraries or that just don't fit into the scope of the application go here. This is useful for libraries shared across applications.
- **`vendor/assets`** - Any assets owned by plugins, etc.

Inside your assets directory you should have a directory for images, javascripts, and stylesheets.

Concatenation

The first, although not the most exciting, feature of the pipeline is concatenation of assets. This is very important to high-traffic production applications as it lowers the number of requests for assets and thus reduces server overhead. Now the default behavior of Rails is to concatenate all assets into a single file per type (so one master Javascript file, and one master CSS document).

To concatenate all the files together, Rails uses a manifest file to determine which files to include. Each of these manifests contain directives telling Sprockets which files to require in order build the single, monolithic document. These directives come in the form of comments in the code, using one of three keywords:

- **require** - includes a single document into the compiled asset output
- **require_self** - Used primarily in CSS, this includes any code in the current file in the compiled asset output
- **require_tree** - Similar to require, this loads *all* the files in the path relative to the current document.

For example, here is the default manifest for Javascripts, located in `app/assets/javascripts/application.js`:

```
//= require jquery
//= require jquery_ujs
//= require_tree .
```

The files for jquery and jquery_ujs are in Sprockets' search path, by being part of the jquery-rails gem included by the Rails application.

For CSS documents, you see the following in `app/assets/stylesheets/application.css`:

```
/*
 *= require_self
 *= require_tree .
 */
```

This loads all CSS documents in `app/assets/stylesheets` after the code in the application.css document itself. Note that the load order listed in the manifest documents is honored, so be careful when listing them.

When the documents are compiled to the single, concatenated form, Rails takes advantage of a concept called fingerprinting to add a hash to the generated filename. So long as the assets do not change, IPSs and browsers can cache the data and serve assets that way. Whenever a change is made to the assets, a new hash is generated, thus invalidating the old cache because an entirely new file will be requested.

So for example, `application.css` can be referenced with a hash like this: application-dd6b1cd38bfa093df600724704fd7a11.css.

Previously, rails appended a query string to the URL based on the file's modified time, so the source looked similar to this:

```
/stylesheets/application.css?201107211050
```

Each time the code is deployed or served from different machines, the potential for not honoring the cache existed. Plus, a number of web servers do not honor the query string and won't cache the content. So it wasn't a fool-proof method of serving up the assets and caching them.

Minification and Compression

The pipeline is also useful for minifying and compressing your asset code. This helps to save bandwidth as your files are transferred to a multitude of users, and it can also help obfuscate your Javascript code to keep the casual observer from benefiting from your hard work.

In CSS documents, this process involves removing comments and unnecessary whitespace.

In Javascript documents this gets to be more elaborate. By default Rails utilizes the uglifier gem, which is a wrapper for the UglifierJS code compressor for NodeJS. By default it will remove whitespace, mangle variable names, "squeeze" code by converting things such as if statements to ternary operations whenever possible, remove dead code that will never be executed, and more. It tightens up Javascript to make it a smaller download but it also serves to obfuscate it from prying eyes.

Preprocessing Languages

The most exciting feature of the asset pipeline has to be the ability to use other languages (or language extensions) that will be compiled down to the native Javascript and CSS code. Now, inside your Rails application you can write CSS using either SCSS or SASS, you can write CoffeeScript for Javascript, or use template languages like ERB in either.

The filenames are similarly structured to how you name views, so you have the filename, followed by the output language (js or css), and then finally use the preprocessor language as your extension. For a CoffeeScript file you would have a file named like this:

```
app/assets/javascripts/traveljournal.js.coffee
```

Or a CSS document like this:

```
app/assets/stylesheets/traveljournal.css.scss
```

One neat feature of the pipeline is the ability to stack processing. You can have a document parsed by ERB and then passed to CoffeeScript by naming it like this:

```
app/assets/javascripts/traveljournal.js.coffee.erb
```

The order is important. Whatever processor you list *last* will be processed *first*. So the above example will be processed by ERB first, then CoffeeScript, finally resulting in the Javascript output.

CoffeeScript

Learning the intricacies of CoffeeScript is beyond the scope of this course, but we can talk a little about what it has to offer.

CoffeeScript looks more like Ruby. You do not need semicolons to end expressions, variables do not need to be pre-declared using `var`, functions are specified with `->` like stabby Procs, and so on.

Some examples of how it looks, taken from http://jashkenas.github.com/coffee-script :

```
# Assignment:
number   = 42
opposite = true

# Conditions:
number = -42 if opposite

# Functions:
square = (x) -> x * x

# Arrays:
list = [1, 2, 3, 4, 5]

# Objects:
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x

# Splats:
race = (winner, runners...) ->
  print winner, runners

# Existence:
alert "I knew it!" if elvis?

# Array comprehensions:
cubes = (math.cube num for num in list)
```

Compiles down to this Javascript:

```
var cubes, list, math, num, number, opposite, race, square;
var __slice = Array.prototype.slice;
number = 42;
opposite = true;
if (opposite) {
  number = -42;
}
square = function(x) {
  return x * x;
};
list = [1, 2, 3, 4, 5];
math = {
  root: Math.sqrt,
  square: square,
  cube: function(x) {
    return x * square(x);
  }
};
race = function() {
  var runners, winner;
  winner = arguments[0], runners = 2 <= arguments.length ? __slice.call(arguments, 1) : [];
  return print(winner, runners);
};
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
cubes = (function() {
  var _i, _len, _results;
  _results = [];
  for (_i = 0, _len = list.length; _i < _len; _i++) {
    num = list[_i];
    _results.push(math.cube(num));
  }
  return _results;
})();
```

SASS

SASS (Syntactically Awesome Stylesheets) was originally part of the Haml gem. It provides extensions for CSS3, such as nested rules, variables, mixins, selector inheritance, and more to your CSS documents. There are two different syntaxes you can take advantage of, but developers are leaning more toward using SCSS (Sassy CSS) as it is a superset of CSS3, meaning that valid CSS3 documents are already SCSS documents. These documents are named with a `.scss` extension.

The other syntax is the original SASS syntax that uses indenting rules similar to Haml as opposed to semicolons and curly braces. It is still supported if your document is named with a `.sass` extension.

Here is an example SCSS document that utilizes variables, mixins, and nesting:

```
$heading: #eee;
$text: #000;

@mixin rounded($radius:10px) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  border-radius: $radius;
}

header {
  @include rounded(5px);
  color: $text;
  background-color: $heading;
  a {
    text-decoration: underline;
    &:hover {
      text-decoration: none;
    }
  }
}
```

This generates CSS output that looks like this:

```
header {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
  color: black;
  background-color: #eeeeee; }
  header a {
    text-decoration: underline; }
    header a:hover {
      text-decoration: none; }
```

In this case, the generated output is smaller, but the SCSS is much more developer friendly as it contains the principles of DRY, reusable code, and better object-oriented concepts than base CSS can offer.

Unit Review

- Views
- ERB
- Haml
- Layouts
- Helpers

Walkthrough 7-1: CoffeeScript

In this walkthrough, we'll transform a routine written in jQuery into CoffeeScript. We'll also look how to let Rails know that this routine should only be served for a particular controller.

1. Before get started with the comparison, we'll see the jQuery function in action. This function is invoked after a new Hotel is saved. It highlights the most recently added hotel for about 10 seconds when the index page is displayed after a new Hotel is created.

   To wire up this functionality, we'll need to use the `id` attribute for each Hotel row in the index screen to let the JavaScript side know when the Hotel was entered into the system. Set the `id` for each `tr` element in `app/views/hotels/index.html.erb` to the integer representation of the value of the `created_at` property for the Hotel and also tag the row with the `hotels` class as follows:

   ```
    <tr class='hotels' id = <%= hotel.created_at.to_i.to_s %>>
   ```

   Add the following hidden field to `app/views/hotels/index.html.erb`. The `HotelsController` will use this field to communicate with the JavaScript routine. The `HotelsController` will let the JavaScript know to highlight the new Hotel by setting this field to "highlight" before rendering the index page. When the index page will not execute the highlighting logic under any other circumstances.

   ```
   <%= hidden_field_tag :highlight, @highlight%>
   ```

   In the `create` action for the `HotelsController`, add a the key/value pair `:highlight => "highlight"` to the `redirect` directive that is invoked if a Hotel is successfully. This will add `:highlight => "highlight"` to the `params` hash accessible to the `index` action:

   ```
   redirect_to :action => 'index', :highlight => "highlight"
   ```

   Now add a single line of code to the `index` action so that the hidden field we added to the view will be populated with the value paired with the `:highlight` key in the `params` hash:

   ```
   @highlight = params[:highlight]
   ```

   The hidden field we added to the form will only have a value when `index` action is invoked via `redirect` from the `create` action.

   Add the following directive to `app/assets/javascripts/application.js` and restart your server to update the paths:

   ```
   //= require jquery-ui
   ```

   Paste the following jQuery routine into `app/assets/javascripts/hotels.js`. If you rename `app/assets/javascripts/hotels.coffee.js`, which was generated when the `HotelsController` was generated -- be sure to remove the pound-sign-prefixed comments at the top of the file. Those comments are valid CoffeeScript, but not valid JavaScript!

   ```
   $(document).ready(function() {
     if ($("#highlight").val() == "highlight") {
         var newest_hotel = $('.hotels')[0]
         $(".hotels").each(function (i) {
   ```

```
          if (this.id != "") {
             if (parseInt(this.id) > parseInt(newest_hotel.id)) {
                 newest_hotel = this;
             }
          }
        });
 //
        if ( $("#" + newest_hotel.id).offset().top >=  $(window).height() ) {
           window.scrollTo( $("#" + newest_hotel.id).offset().left,
                            $("#" + newest_hotel.id).offset().top);
        }
 //
     $("#" + newest_hotel.id).effect("highlight",{},9000);
     }
});
```

Now try adding a new Hotel. The web app should bring you to the index page, and the newly-added Hotel should be highlighted. If necessary, the web app should scroll down to make the new Hotel visible.

2. Here's the equivalent CoffeeScript:

```
$(document).ready ->
  if $("#highlight").val() is "highlight"
    newest_hotel = $('.hotels')[0]
          for hotel in $('.hotels')
            if (parseInt(hotel.id) > parseInt(newest_hotel.id))
              newest_hotel=hotel unless hotel.id is ""
 #
          if ($("##{newest_hotel.id}").offset().top) >= ( $(window).height() )
            window.scrollTo( $("##{newest_hotel.id}").offset().left ,$("##{newest_hotel.id}").offset().top)
 #
          $("##{newest_hotel.id}").effect("highlight",{},9000)
```

Some points to consider:

- In jQuery, looping through all the elements selected by the `$('.hotels')` selector to determine which has an `id` that represents the most recent date\time requires an anonymous function replete with the `function` keyword & curly braces, while in CoffeeScript, the logic to execute for each element in the collection can be expressed by simply indenting.
- How optional parentheses work nicely with some of the CoffeeScript's natural language features, like support for `is` and `unless`. For example, compare this jQuery snippet ...

  if (this.id != "") {

  }

... to the CoffeeScript translation:

```
    unless hotel.id is ""
```

```
* String concatenation in jQuery ('$("#" + newest_hotel.id)') vs. string interpolation in CoffeeScript ('$("##{newest_hotel.id}")')

* The '});'formation that turns up in JavaScript when functions are used as function arguments
```

1. Try replacing the jQuery with CoffeeScript. Rename `app/assets/javascripts/hotels.js` to `app/assets/javascripts/hotels.js.coffee` and paste in the CoffeeScript version of this routine. Try it out.

2. Take a look at the generated JavaScript, by navigating to `http://localhost:3000/assets/application.js`. The JavaScript in this file is available to every screen, yet the function, as written is very Hotel-centric. The function could certainly be written in a more generic way, and could potentially be useful for a variety of models, but for now, we're going to use it to represent logic that should only be loaded for a particular screen. There are several ways to do this. The `//= require_tree .` indicates that all js under `assets/javascripts` should be loaded. You can use `require` to specify that only particular files should always be accessible or you could place controller-specific JS files in a directory named `restricted` or `custom` or `models` and use `//= require_directory` to indicate that only files directly under `app/assets/javascripts` should be loaded. Try one of these methods. You can use `<%= javascript_include_tag "hotels" %>` in `app/views/hotels/index.html.erb` to load the hotel highlighter logic for the index page exclusively.

Walkthrough 7-2: Unobtrusive JavaScript

This walkthrough will give you some experience with unobtrusive JavaScript. We'll use a form that is not backed by a model for this exercise, so you can try out the `tag` versions of the view helpers: It's a form that captures sort parameters on the hotels index page and reorders the list of hotels.

This form leverages the `form_tag` helper to generate an Ajax-powered form when Javascript is available. We are going to get the form up and running without Javascript support, and then we'll enhance it with Javascript.

Steps

1. Add the following form to `app/views/hotels/index.html.erb` below the "as of" banner:

   ```
   <%= form_tag hotels_path, :method => :get , :remote=> true  do%>
     <fieldset style="width:40%;">
       <%= radio_button_tag("sort_column", "name", true) %>Name
       <%= radio_button_tag("sort_column", "description") %>Description
       <%= submit_tag "Sort" %>
     </fieldset>
   <% end %>
   ```

   The first argument to `form_tag` is the URL that points to the action that will be invoked when the form is submitted. In this case, the form `hotels#index` action will be invoked. The default "method" for a form generated via `form_tag` is POST, but you can specify an alternative. We are going to modify the `index` action to use the specified sort order when the user submits a sort request and to use a default sort order otherwise.

   The `radio_button_tag` helpers generate a radio group used to indicate whether the list of hotels should be sorted by "name" or "description". The first argument specifies the name of the key that will represent the radio button in the `params` hash and the second argument represents the value that will be paired with that key if the radio is selected. Two or more radio buttons with the same hash key constitute a group of choices. Only one of these choices can be selected when a form is submitted. When the user clicks on one radio button in a group, the other radios in that group are automatically deselected.

The third argument to `radio_button_tag` specifies whether the radio button should be selected when the form first loads. It is false by default. We're setting this argument to true for "name", marking it as the default "Sort By" column.

Note that the `radio_button_tag` helper does not generate a labels. In this form, plain text is used to label these widgets.

2. In this step we'll examine the `params` hash that gets dispatched to the web app when the sort form is submitted.

   Navigate to the hotels index page (`/hotels`) and hit the "Sort" button. Take a look at the resulting `params` hash in the server log. It should look something like this:

   ```
   Parameters: {"utf8"=>"â", "sort_column"=>"name", "commit"=>"Sort"}
   ```

   Now click the "Description" option. Submit the form and view the server log. Now the "sort_column" value should be "description."

3. Next we'll inspect and use the `params` hash in the `hotels#index` action.

   Currently the web app is hard-wired to sort the hotels by name. We'll use the form values to build an "order by" clause dynamically. We'll encapsulate the logic that builds the "order by" clause in the `order_by_clause` method. Add the following method to `hotels_controller.rb`:

   ```
   def order_by_clause
     if params[:commit] == "Sort"
       params[:sort_column]
     else
       "name"
     end
   end
   ```

   This method determines whether the action was invoked via the Sort form by checking to see whether the `commit` key value in the `params` hash is "Sort". If the form was submitted, the "order by" clause is built up using the form values -- otherwise, "name" is used as a default order_by clause. In the alternative, the developer could use `request.post?`, which returns true if the request was a POST request. When the hotels list is accessed outside of the form, it will be via a GET request.

   To complete the 'non-Javascript' `sort` implementation, replace the text `name` as the argument to `Hotel.order` in the `index` action with a call to `order_by_clause` as follows:

   ```
   Hotel.order(order_by_clause).
   ```

   To simulate an environment that lacks Javascript support, either disable Javascript in your browser or comment out the `require` statements in your `app/assets/javascript/application.js` file and restart the Rails server. You can comment out the `require` statements by removing the equal sign to the left of each `require`.

   Test the sort form.

   In the server log, take a look at the line above the `Parameters` hash. You should see something like this, indicating the an HTML request has been made:

```
Processing by HotelsController#index as HTML
```

When you are done, don't forget to reactivate JavaScript in your browser or restore the removed equal signs. To make sure JavaScript has been enabled, press the `sort` button again. Sorting will not work, because we have yet to implement a response for a JavaScript request -- but you should see a line like the following, indicating that a JS request has been made:

```
Processing by HotelsController#index as JS
```

4. Now we can enhance the form with Ajax. The `form_tag` helper generates a form tag with its `data-remote` attribute set to "true" if you include `remote => true`. As you just saw, the `data-remote` attribute caused no problems when we tested the form sans JavaScript.

   When you start up an application with JavaScript enabled, the jQuery script that is installed by default when you use `new` to generate a Rails app looks for links, forms, buttons and other widgets tagged with `data-remote`. It dynamically AJAX-enables those elements.

   When you submit the form with Javascript enabled a request is issued that expects to get JavaScript back.

   We have talked about how Rails handles xml, HTML and JSON. Soon we'll experiment with the Rails framework's support for Javascript.

   When an HTML response is requested, Rails expects to find a file with the same name as the action in a directory under `app/views` as the first component as the filename, followed by the 'html', followed by the name of the default templating language (ie 'erb' or 'haml'). An example would be `index.html.erb`.

   Similarly if a JS response is requested and your code doesn't explicitly reference a template Rails will look for a file with a name like `index.js.erb`.

   So...

4a. Let's create `index.js.erb` in `app/views/hotels`. For now, let's just put some stub JavaScript in there. Enter an alert: `alert("YO!");`

4b. Modify the `respond_to` call at the top of `hotels_controller.rb` to reference JavaScript as follows: `respond_to :html, :xml, :json, :js`

1. We'll still need to a little bit of work before we can see the JavaScript-enhanced sort in action, but we are at good stopping point to try submitting a sort request to verify that the logic in `index.js.erb` will get invoked when the sort form is submitted. Go ahead and view the hotels index screen (`\hotels`) and click on the "Sort" button.

2. Next we will enclose the portion of the page that we want to redisplay after pressing the Sort button within `div` tags and identify the `div` with an id, as follows (we'll use the id "hotels"):

```
<div id="hotels">
  <table>
    <tr>
      <th>Name</th>
      <th>Description</th>
    </tr>
      <% @hotels.each do |hotel| %>
        <tr>
          <td><%= hotel.name %></td>
          <%= description_column(hotel.description) %>
        </tr>
      <% end %>
  </table>
</div>
```

The id enables us to easily manipulate the hotels div with jQuery. Before we add code to display the sorted hotels, we'll try out one of the many effects jQuery supports. Replace the `alert` call in `index.js.erb` with `$('#hotels').fadeOut();`

You can use CSS-like syntax as an element filter in jQuery. We're using the pound sign to specify that we're looking for an element with the id of "hotels". If you wanted to select elements tagged with a class called "hotels", you'd use ".hotels".

Load the hotels index and try the Sort button.

3. Next we should put the actual list of hotels in it's own file so it will be easier for Rails to just reference the one piece of the file that needs to be displayed. Cut out the following code and place it in a new file called 'app/views/_hotels.html.erb'.

   View fragments are referred to as "partials" in Rails. The naming convention for partials is to prefix their names with an underscore.

   In order to display a partial, you pass it to the `render` method. To display the hotels partial in `index.html.erb`, insert the following text in the same spot in the file where the list of hotels initially lived: `render "hotels"`. Note that when you pass the name of a partial to the `render` method, you need to omit the underscore.

   Lastly replace the `fadeOut()` call with the following line:

   ```
   $("#hotels").html("<%= escape_javascript(render('hotels'))%>");
   ```

   This line of code will replace the HTML inside the div with an id of "hotels" with the output of rendering the "hotels" partial. Note the `erb' tags mixed with the JavaScript. You don't have free reign withinerb`' tags when you mix them into JavaScript in the same file, but you can do some powerful things. Hat tip to Ryan Bates for demonstrating this cool technique in his Railscast about unobtrusive JavaScript and Rails 3.1: http://railscasts.com/episodes/205-unobtrusive-javascript.

   Try out the sort button, now that the sort feature is fully implemented.