Basic Ruby

Starting Ruby

Hello and Welcome to this super exciting little introduction to Ruby, the language that aims to make developers fall in love with programming again. Ruby is a super expressive, open language that lets you do an awful lot in next to no time at all.

It's the language that powers most of the world's tech startups. Why? Because it lets small teams of crack developers build incredible web applications that would have taken months, or even years using traditional techniques.

Wow your clients with your awesome productivity. Tackle side projects with ambition and alacrity.

Feeling excited yet?

Here are some of the great things about Ruby:

- Very high level - a little bit goes a long way.
- Super light, clean, flexible, some would say poetic syntax.
- Genuinely fully object oriented. Integers are objects. Even methods are objects.
- Introspection is built in, not tacked on.
- Operators overloading is super simple. Define your own types and do maths with them.
- Not written a method yet? With MethodMissing, Ruby can write it for you, while your code is running.
- Interpreted, so no build, compile, debug cycle to monkey up your flow.
- Sensible conventions.
- Awesome frameworks: Rails, Sinatra and more.
- Massive library of Gems to call on. You rarely have to reinvent the wheel.
- Super charming

…And more nice little touches than you can shake a stick at.

Now are you excited? You're about to learn a language, designed specifically to make programmers happy, that doesn't make you jump through hoops, and that gives you the respect you deserve. Welcome to Ruby!

Environment

To get started you're going to need some basics. I'm going to assume here that you have a copy of Ruby, a terminal, and a text editor of some description. You'll need access to irb, the interactive ruby interpreter.

The following things need to be true

You can get up a command line or terminal of some description. If you type ruby at the command line, this needs to work and not throw an error If you type irb at a command line, this too needs to work (type exit to get back out again)

If these things are not all true for you, please skip on to appendix 1.

Hello IRB

IRB is the interactive ruby interpreter. If you have access to a command line and a copy of Ruby, you will have IRB. It's a testbed. It lets you try ideas out quickly. Here we're going to use it to write our first Ruby program.

At a command line, type

```
irb
```

to enter interactive Ruby. Got it up? Good. Now type:

```
puts "Hello Ruby you charming little thing"
```

Now press enter. See what you did there? Puts stands for put string. It just appends a string to the current output stream, which here is the terminal. Notice as well what you didn't do. You typed no semi-colons and no braces.

Now lets go a step further

In IRB, type:

```
puts "Hello Ruby " * 5
```

See what happened? Multiplication works on strings. This might seem odd at first, but it's actually consistent with the way that Ruby deals with operators and methods. We'll come to this later.

IRB is tremendously useful for trying things out. Because Ruby is so expressive, you can do a huge amount just on one line. Ruby loves it when you write clean, concise, and above all, expressive code, and IRB can help you do this. You can tackle many of the exercises in this book using IRB.

IRB Exercise

Throughout this book you'll find lots fun and exciting exercises. I'd encourage you to try them out, but be pragmatic. If something is really simple and obvious, you can skip it, I won't mind. Here's a simple one for starters:

1. Get a hello world application running in IRB.
2. Make it print Hello World 50 times, 1000 times, 100000 times.

Answers

IRB Exercise

Open IRB by typing IRB. Then:

```
puts "Hello Ruby " * 50
puts "Hello Ruby " * 1000
puts "Hello Ruby " * 100000
```

Basic Data Types

In this section we'll look at variables and the basic data types that you'll find in pretty much any language: strings, integers, floats, Booleans and arrays. You might be tempted to skip this section. Don't. Ruby's "everything is an object" philosophy means that you can do a lot more with basic types than is possible in other languages.

Variables

Any discussion of basic types should start with an introduction to variables. Ruby is very lax about variables, it cuts you a lot of slack and assumes for the most part that you know what you are doing.

Variables are duck typed

Variables in Ruby are **duck typed**. If they can quack like a duck, they will be allowed to swim on the pond. The interpreter will not do any type checking for you in advance. This might upset people coming from a .Net, C++ or Java background, and it's one of the reasons Ruby is suited to small teams of crack developers, since you can't enforce an interface, and can't prevent people from passing silly parameters to your methods.

It's also one of the reasons that Ruby is so fabulously productive, since you don't need to interact with the type system. Polymorphism is assumed. You don't need to do any work to enable it. Productivity wise, this is an enormous win, provided you can trust yourself and your co-workers.

Declaring variables

Variables come into existence when they are first declared. There is no need to define them.

For example:

```
hi = "Hello Ruby"
a_big_number = 1000
```

This is pretty sensible. A variable can hold anything you like, and the same variable can be repurposed to hold something else entirely:

```
a = 10
a = "red"
```

A little best practice: naming conventions

There are naming conventions governing variable names in Ruby. These conventions are not enforced, but you should stick by them if you want people to like you, since Ruby is case sensitive.

Variable names

Variable names always start with a lower case letter. By convention they are all lower case with optional underscores (snake case) eg:

```
number_of_people
user_name
height_of_the_eiffel_tower
```

Constants

Constants start with an upper case letter and by convention are CAPITALISED_SNAKE_CASE:

```
MAX_NUMBER_OF_PEOPLE = 20
NUMBER_OF_DAYS_IN_A_LEAP_YEAR = 364
```

Note that constants are not actually constant, you can redefine them if you really, really need to. You'll get a warning, but it won't break. Ruby is like this, it assumes you're clever. Yes, Ruby is a language that gives you the respect you DESERVE.

Clever Tricks

There are lots of little tricks you can do with variables that are useful, and can help a lot when trying to appear clever.

Assignment chaining

Assignments can be chained saving typing eg:

```
x = y = z = 4
puts x + y + z
  => 12
```

This works because the output of the assignment is the value that is being defined, so the output of z = 4 is 4.

Parallel assignment

Ruby also supports parallel assignment allowing you to assign multiple different variables on one line, eg:

```
a,b = 5,6

a
  => 5
b
  => 6
```

You can exploit this to swap the values of two variables in one line:

```
a,b = b,a

a
  => 6
b
  => 5
```

Strings

Making Strings

String creation is similar to other languages

```
string_1 = "Hello Ruby"
string_2 = 'Hello Everyone'
puts string_1
  => Hello Ruby
```

```
puts string_2
  => Hello Everyone
```

You are free to use single or double quotes around your string.

Strings are Objects

Strings are real objects and have methods:

```
s = "Hello Ruby"
puts s.reverse
    => "ybuR olleH"
```

String Literals are allowed

As you might expect, you can create strings, and call methods on them directly.

```
"Hello Ruby".upcase
  => "HELLO RUBY"
```

Embedding Variables in Strings

Use double quotes and #{} syntax if you want Ruby to look for variables in a string:

```
name="Derrick"
puts "Hello, my name is #{name}"
  => "Hello, my name us Derrick"
```

Nice, simple, inline, readable.

Escaping with \

You can include escape characters in your string with a backslash:

```
"Ruby's great! \n Oh yes it is!"
  => Ruby's great
  => Oh yes it is!
```

String concatenation

Add two strings together using simple arithmetic:

```
"Hello " + "Ruby"
  => "Hello Ruby"
"Hello " << "Everybody"
  => "Hello Everybody"
"Hello " * 2
  => "Hello Hello"
```

Conversion to other types

Strings can be converted to lots of other types usingthe casting methods. There are lots of these built in, but feel free to write your own as well. The “to_i” method converts to an integer.

```
"5".to_i
  => 5
```

```
"99 Red Balloons".to_i
  => 99
```

Conversion from other types

Strings can be created from any other type using the to_s method. For example, if you have an integer, and you need it to be a string, do it like this:

```
5.to_s
  => "5"
```

Working with the String API Exercise

Visit the String api documentation on Ruby-doc.org and attempt the following

1. Create a string and assign it to a variable.
2. Reverse the string using the reverse method.
3. Permanently reverse the string using the reverse! method. Methods that end in an ! are destructive, they affect the original variable.
4. Create a variable which contains your name. Now use the #{} syntax to create a greeting string that incorporates this variable.
5. Investigate gsub. Use it to replace a portion of your string.
6. If you're into regular expressions, use gsub to replace all instances of a substring with a new substring.

Integers (Fixnums)

Integers are Objects Too

Everything in Ruby is an object including integers, or Fixnums as they are known in Ruby. Integers therefore have methods.

```
x = 1
x.to_s
  => '1'
```

Basic Maths

You can do all the maths you would like with integers. It all works as you would expect:

```
1 + 2
  => 3

5 - 1
  => 4

3 * 4
  => 12

4 / 2
  => 2
```

Integer Shortcuts

You also have the common maths shortcuts you find in other languages. Again, these work as you would expect:

```
x = 1
  => 1

x += 5
  => 7

x *= 2
  => 14
```

Exponentiation

Use ** for exponentiation should you feel that way inclined:

```
2 ** 2
  => 4
a ** 3
  => 8
a ** 4
  => 16
```

There is no Incrementation operator

A surprise here. Ruby has no pre/post increment/decrement operator. For instance, x++ or x– will fail to parse. Use x += 1 instead. There are some relatively technical and quite good reasons for this to do with consistency.

Comparison

Use greater than or less than signs to do comparison. == tests for equality.

```
1 > 2
  => false

2 >= 2
  => true

1 < 2
  => true

1 == 1
  => true
```

The Spaceship Operator

A tremendously useful one this, the spaceship operator returns 1, 0 or–1 depending on whether the first operand is larger, the same as or smaller than the second. This is excellent, as this type of comparison forms the basis of all known sorting algorithms.

To make any class sortable, all you need to do is to implement the spaceship operator on it. Telling a custom class how to respond to an operator is called operator overloading. More on operator overloading later.

```
4 <=> 3
```

```
  => 1

4 <=> 4
  => 0

2 <=> 10
  => -1
```

Large integers

Underscores can be included in integers to make them more legible. We don’t do this often, but it’s a fun trick to know:

```
x = 100_000_000
y= 100000000
puts x == y
  => true
```

Looping with Integers

Remember how everything in Ruby is an object? This lets us do some reasonably clever things. For example, we can it to perform looping, like so:

```
5.times {puts "hello"}

hello
hello
hello
hello
hello
```

This might look strange. The little bit of code between the curly brackets {puts “hello”} is called a block. We are actually passing this block to the Fixnum times method, which is then executing it 5 times. This might seem a little backwards, but it’s actually really good, as we shall see when we reach the section on blocks.

Floats

Ruby also has floating point numbers. Floats are useful for dealing with very large numbers

Declaring floats

Declare a float just by using a decimal point, like so

```
a = 0.5
  => 0.5

a = 1.0
  => 1.0
```

Converting to a float

You can convert integers to floats using the to_f method like so:

```
15.to_f
```

```
  => 15.0
```

Integers are not implicitly converted to floats, so:

```
3 / 2
  => 1
```

rather than 1.5

However, if one of the operands is already a float, the output will be a float so:

```
3.0 / 2
  => 1.5

3 / 2.0
  => 1.5
```

Infinity

Floats have the rather handy ability of extending to infinity, like so:

```
1.0 / 0
  => Infinity

(1.0 / 0).infinite?
  => 1

(-1.0 / 0).infinite?
  => -1
```

If you need to say that something, say a stack, can contain an unlimited number of values, you can just use infinity.

Arrays

Like most languages Ruby allows you to create Arrays to hold multiple values. Arrays in Ruby are one dimensional, though you can declare an array of arrays if you feel the need.

Creating Arrays

Create an array using square brackets like so:

```
cakes = ["florentine", "lemon drizzle", "jaffa"]
```

Access an array using square brackets:

```
cakes[0]
  => "florentine"
```

Zero Indexed

Arrays are zero indexed, so 0 is the first element.

Polymorphic

As you might expect from Ruby, arrays can hold any type of element. This is allowed:

```
["hello", 1, 0.5, false]
```

Manipulating arrays

Arrays can be manipulated in a huge variety of ways For example:

Adding

```
[1,2,3] + [4,5,6]
 => [1, 2, 3, 4, 5, 6]
```

Subtracting

```
[1,2,3] - [1,3]
 => [2]
```

Appending

```
[1,2,3] << 4
 => [1, 2, 3, 4]
```

Multiplication

```
[1,2,3] * 2
 => [1, 2, 3, 1, 2, 3]
```

Handy dandy array methods

There are also a raft of array methods we can use

```
[1,2,3].reverse
  => [3, 2, 1]

[1,2,3].include? 2
  => true
```

You might notice here that this method name has a question mark in it? Methods with a question mark return true or false.

```
[4,9,1].sort
  => [1, 4, 9]
```

Array splitting

Parallel assignment works when pulling values out of a array.

```
array_of_numbers = [1,2,3,4]
a,b = array_of_numbers
a
  => 1
b
  => 2
```

We can also pull the first element, and return the rest of the array should we wish to:

```
arr = [1,2,3]
a,*arr = arr
a
  => 1
arr
  => [2, 3]
```

This is a clever trick to know as it impresses people and make you look brainy.

Creating an array using the to_a method

The to_a method allows many objects to be converted to arrays. For example an array can be created from a range as follows

```
(1..10).to_a
  => [1,2,3,4,5,6,7,8,9,10]
```

Ranges are a Ruby type that allows you to represent a sequence. More on Ranges shortly.

Booleans, Logic, nil and undefined

What language would be complete without Booleans? Ruby supports them rather nicely.

What counts as false?

Only Nil and False are false in Ruby. If it exists it’s true. That includes zeros, empty strings, etc.

We can test this with a short function, that determines if the parameter evaluates to true or false, like so:

```
def true?(value)
  if (value)
    true
  else
    false
  end
end

true?(false)
  => false

true?(nil)
  => false

true?(0)
  => true

true?(true)
  => true

true?(15)
  => true

true?([0,1,2])
  => true

true?('a'..'z')
  => true

true?("pears")
  => true

true?(:bananas)
  => true
```

A variable is nil if the variable has been declared but doesn't point to anything (remember Ruby is fully object oriented so all variables are pointers to objects)

Nil

A variable is nil if it has been declared, but holds no value. Nil exists as a type, and has methods. For example:

```
a = nil
a.to_s
  => ""
a.nil?
  => true
```

Nil is a very useful thing to be able to return. It means "no value".

Undefined

A variable is undefined if it has not been declared. You can test for this using the defined? operator.

```
a = 1
defined? a
  => "local-variable"
defined? b
  => nil
```

Boolean Algebra

You can do all the standard things using Boolean algebra. &&, || and == are all supported.

Special uses for logical OR ||

There are useful things that can be done with the OR || command. The second part is only evaluated if the first part returns false (nil or false evaluate to false), and the return value is the last value calculated. Rails exploits this letting you do neat things like this:

```
name = nil
user_name = name || "Anonymous Coward"
```

Here we have a default value. If name is nil, anonymous coward will be used instead.

Flow Control

Ruby loves it when you tell it what to do.

if/elsif/else/end

If statements are present. They work as you'd expect. Notice that no braces are required.

```
bacon = true
fish = false
if fish
  puts 'I like fish'
elsif bacon
  puts 'I like bacon'
else
```

```
  puts "I don't like fish or bacon"
end

  => 'I like bacon'
```

unless

The unless keyword is the opposite of the if keyword. It’s equivalent to !if (not if). It can make your code more readable.

```
bacon = false
unless bacon
  puts 'fish'
else  puts 'bacon'
end

  => "fish"
```

All on one line?

All on one line?

Ruby loves it when you write things concisely. The if and unless keywords can also be placed after the line of code you may or may not want to execute. When used in this way they are called statement modifiers:

```
user = "dave"

puts "Hello #{user}" if user
puts 'please log in' unless user
```

The Ternary operator

The Ternary operator

Like most languages Ruby includes a ternary operator. It works as you’d expect. If the first portion evaluates to true the first result is given, otherwise the second result is given:

```
bacon = true
puts bacon ? 'bacon' : 'fish'
  => "bacon"
```

This is equivalent to:

```
bacon = true
if bacon
  puts 'bacon'
else
  puts 'fish'
end

  => "bacon"
```

Case Statements

Of course Ruby also has case statements. You almost never see these in the wild as case statements are a bit, well, 1990s, but if you should decide you need one, here's how they work.

```
refreshment_type_required = "biscuit"
suggest_you_eat = case refreshment_type_required
  when "pastry" : "cinnamon danish whirl"
  when "hot drink" : "mocha with sprinkles"
  when "biscuit" : "packet of bourbon creams"
  else "glass of water"
end
```

Loops

Ruby is unusual in it's approach to looping. It has all the usual constructs, for, while and for in, but for the most part we perform iteration by passing blocks around.

However Ruby does have most of the usual looping constructs in case you ever need them, and has very clean syntax.

While Loop

```
bacon = 0
while bacon < 10
  puts "bacon is limited to #{bacon += 1}"
end
puts "my bacon collection is now complete"
```

While can operate on a block. This is equivalent to do while.

```
begin
  bacon += 1
end while bacon < 10
```

While as a Statement Modifier

A while loop can also be used as a statement modifier. We can repeat a single line until a condition is true.

```
amount_of_bacon = 0
desired_amount_of_bacon = 10_000

amount_of_bacon += 10 while amount_of_bacon < desired_amount_of_bacon
puts "I now have #{amount_of_bacon} bacon"
```

Until Loop

An until loop is the opposite of a while loop. The code will be iterated over until a condition becomes true.

```
cheese = 0
until cheese >= 10
  puts "cheese level is insufficient: #{cheese += 1}"
end
puts "we have sufficient cheese to proceed"
```

Until as a statement modifier

We can also use until as a statement modifier, looping over a single line until a condition becomes true.

```
puts "Cheese level increasing to: #{cheese += 19}" until cheese >= 150
```

For loop

Ruby for loops are a little different from many other languages. They always iterate over an object, be that a range or an array, like so:

Iterating over a range:

```
for i in 1..5
  puts i
end
  => 1
  2
  3
  4
  5
```

Iterating over an array:

```
for i in ['hats','scarves','gloves']
  puts i
end
  => hats
  scarves
  gloves
```

High / Low Exercise

We don't use loops in Ruby nearly as often as in other languages, since blocks provide most of same functionality with a nicer syntax and improved encapsulation, but we do still occasionally need them.

1. Write a program that allows the user to repeatedly enter a number and tells them if it is too high or too low. Exit when they guess correctly.
2. Re-implement the last task using an until loop.

Hashes and Symbols

Hashes are a big deal in Ruby. We use them a lot, particularly when passing bits of data around. Symbols are tiny lightweight Ruby placeholder objects. They are often used in conjunction with hashes. In this section we will look at hashes, symbols, and some of their common uses.

Symbols are not Magic Runes

Symbols are not magic runes

Symbols often seem like magic runes to Ruby newcomers. In fact they're just little objects that have some special features and syntax surrounding them to make them a little easier to use and a little lighter on the computers memory.

Symbol syntax

Symbols are defined using the colon operator or the to_sym method:

```
:price
:length
:outrageousness
"My Price".to_sym
```

Features of Symbols

- Symbols are tiny little objects
- Symbols don’t have a value, they are placeholders, not variables.
- There can only ever be one symbol with a particular name, this is managed by Ruby. Because of this they are not processor or memory intensive.

There can only be one

A symbol will only exist once in memory, no matter how many times it is used. If for example, you create two symbols in different places both called :name for example, only one object would be created. This object would persist for as long as the Ruby interpreter was running.

Uses of Symbols

Symbols are most commonly used as placeholders in Hashes. We have used symbols already in Rails, for example the Rails params hash associates any the values of any parameters passed in by the user (from a form or in the url) with a symbol representing the name of that value.

Hashes

Hashes are objects that associate lists of arbitrary objects with each other. For example:

```
animals = Hash.new
animals[:tall] = "giraffe"
animals[:minute] = "kitten"
puts animals.inspect
```

Hash Assignment Shorthand

```
animals = {:tall => "giraffe", :minute => "kitten"}
puts animals[: minute]
  => kitten
```

Setting Default Values in a Hash

A hash will return nil if it can’t find any matching keys. You can set a default value that a hash will return in this instance should you so desire.

```
animals = Hash.new("monkey")
puts animals[:funny]
  => "monkey"
```

You can also set this value using the Hash.default method

```
animals.default = "star mole"
puts animals[:odd]=> star mole
```

Any Objects can be used as a key

Any Object can be used as a key. Here we use the number 45 as a key, and store the root directory of the local file system as a value:

```
animals[45] = Dir.new '/'
puts animals.inspect # {45 => #<Dir:0x284a394>
```

It’s all allowed.

One caveat to bear in mind, if you are using Ruby 1.8.6 or less, you can’t use a hash as a key in another hash. This is an issue that was addressed in Ruby 1.8.7 and up.

Passing hashes to a method

Ruby has a useful shorthand for passing a hash to a method. As long as no arguments come after the hash you can forget about the braces giving us a nice rails style way of talking to our objects. Rails uses this syntax extensively.

More on this in the section on Functions.

Functions

Defining Functions

Functions are declared using the def keyword:

```
def greeting
  puts "Hello Ruby"
end

greeting()
  => Hello Ruby
```

Accepting parameters

Functions can accept parameters as you would expect. We pass them like this:

```
def greet(name)
  puts "hello #{name}"
end

greet("dave")
=> "hello dave"
```

Optional braces

When calling a function, the braces are optional.

```
greet "dave"
```

```
  => "hello dave"
```

This is a really nice syntax, and comes into it’s own when we start writing methods.

Returning a Value

Functions can return a value. We pass back a value using the return statement, like so:

```
def get_greeting_for(name)
  return "hello #{name}"
end

puts get_greeting_for "dave"
=> "hello dave"
```

get_greeting_for “dave” **evaluates** to the string “hello dave”. This string is received by puts, which then outputs it to the screen.

Optional return statements

The return statement is also optional. If it’s omitted the function will return the last evaluated expression, so:

def get_greeting_for(name) “hello #{name}” end

```
puts get_greeting_for "dave"
=> "hello dave"
```

This is a clean and useful syntax for short methods such as getters.

Default Values

We can set the default value of an argument, so if no value is passed, our function will still work:

```
def get_greeting_for(name="anonymous")
  return "hello #{name}"
end

puts get_greeting_for
=> "hello anonymous"
```

Note that if we have several arguments, and some are missing, they will be filled in from left to right, so the last ones will take their default values.

Receiving a Hash

A function can receive a hash of values. This is tremendously useful, and we do this all the time.

```
def get_greeting_for(args={})
  name = args[:name] || "anonymous"
  return "hello #{name}"
end

puts get_greeting_for :name => "Fat Tony"
=> "hello Fat Tony"
```

```
puts get_greeting_for
=> "hello anonymous"
```

Here our function receives a parameter we've called args. The default value of the args parameter is an empty hash. Any key value pairs we pass will go into args, and can be pulled out.

On the second line we do this:

```
name = args[:name] || "anonymous"
```

Here we set the value of name to be either the value stored in args under the :name key, or if this evaluates to nil (and therefore false) we set it to anonymous. This is tremendously useful, since we can create functions that accept multiple arguments, in any order, with any defaults that make sense.

You should get used to writing configurable, extensible methods that receive a hash. This is a real rubyism.

Ranges

Ranges are useful objects that can be used to represent a sequence. Ranges are defined using the .. or … syntax. For example

```
1..10
```

represents the sequence of numbers between 1 and 10.

```
1...10
```

represents a range that excludes the high value. In this case the numbers 1 to 9

Ranges in Memory

Ranges are compact. Every value in the range is not held in memory, so for example the range:

```
1..100000000
```

…takes up the same amount of memory as the range:

```
1..2
```

Conversion to arrays

Ranges can be converted to arrays using to_a like so

```
(1..10).to_a
  => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note we need to put braces around the range here to disambiguate the dots.

Character Arrays

We can also declare arrays of characters like this:

```
'a'..'f'
```

Testing if a range contains a value.

We can use the equality === operator to test if a range contains a value. For example:

```
('a'..'f') === 'e'
  => true

('a'..'f') === 'z'
  => false
```

Objects

Ruby is Object Oriented. Many languages claim to be object oriented, but most fall short in some area or another. When Ruby says it's object oriented, it really, really means it.

We have used lots of Ruby objects so far, when we have done things like

```
"hello".reverse
```

and

```
5.upto 1000
```

Lets look now at how we can define our own objects.

Classes

Ruby has a class based object model. This means we define classes, then use them to stamp out as many objects as we like. Of course, this being Ruby, classes are objects and have methods in their own right, but we'll get to this soon enough.

Lets look now at defining a class.

```
class Pet
end
```

Here we have defined a very simple class, called Pet. Classes in Ruby always start with a capital letter. Now lets create a pet:

```
flopsy = Pet.new
```

This is great, we have created a new instance of the Pet class. Flopsy is an instance of Pet.

Note that all objects get a new method for free. This they inherit from the Object superclass. More on inheritance in a bit.

Finding the class of an object

We can go in reverse, to find the class of an instance like this

```
flopsy.class
  => Pet
```

Ruby doesn’t shy away from introspection, it comes baked in, as we shall see later.

Methods - Giving Flopsy Abilities

This is all very nice, but flopsy is not very interesting, she can’t walk the tightrope or play chess, or really do anything much. To make Flopsy more interesting, we need a method:

```
class Pet
  def play_chess
    puts "Now playing chess"
  end
end
```

Here see now. We have added a play chess method to flopsy. We can now write:

```
flopsy.play_chess
```

…and she will, after a fashion. She is only a housepet after all.

Naming Conventions

There are a few things to bear in mind when naming methods in Ruby if you want to appear cool and down with the kids.

First, use snake case for all function names, like this.

```
each_with_index
```

Second, if your method returns a boolean, and is a question, frame it as such. Use a question mark, like so:

```
['toast','jam','honey'].include? 'ham'
person.has_name?
password.valid?
```

Third, if your method modifies the original object in place, rather than returning a new object, and is therefore destructive, indicate this with an exclamation mark, like so:

```
['toast',['jam','honey']].flatten!
  => ['toast','jam','honey']
```

Instance variables - Giving flopsy some superpowers

Flopsy is still a little dull. It would be great to be able to store some data about her, maybe give her some custom attributes.

In Ruby we save an instance variable using the @ syntax. Instance variables are @ variables. All instance variables are private, so to get at them, we need to write methods called getters and setters to access them. Lets have a look now:

```
class Pet

  def super_powers=(powers)
    @super_powers = powers
  end

  def super_powers
    @super_powers
  end

end
```

Here we have given flopsy two methods, a getter and a setter. The first is a setter. The super_powers= method receives a parameter and stores it in an instance variable called @super_powers.

The second is a getter. It simply returns the @super_powers instance variable that was previously set.

We can now set flopsy's super power like this:

```
flopsy.super_powers = "Flight"
```

and retrieve it like this:

```
flopsy.super_powers
  => "Flight"
```

Note we don't have to declare the @super_powers variable anywhere. We can just set it, and that's fine.

This is great because…

Getters and setters give us a clean way to provide an interface onto our object. It insulates us from implementation details. We are free to store the data in any way we wish, as a variable, as a file, in a database, in an encrypted hash, or as a combination of other variables.

This is how active record works when using Rails. Values can be got from the database as though we were accessing object attributes.

Just like Flopsy, the boundary between attributes and methods is far more fuzzy than in most languages. This is partly because of Ruby's optional parentheses, which make it look as though we are accessing attributes, when in fact we are always accessing methods.

We can have read only attributes by only creating a getter, and write only attributes by only creating a setter. An example of a write only attribute would be a password, which might get set, and then encrypted with a one way hash, never to be read again.

The attr method

Since class variables are so common, ruby defines shortcuts for creating them and their associated getters and setters. The attr method creates an attribute and its associated getter. If the second parameter is true a setter is created too. The attr_reader and attr_writer methods create getters and setters independently.

Initialising flopsy using the initialize method.

The initialize method is called by the new method and it is here that we put any code we need to initialize the object.

When flopsy's sidekick mopsy was first created, she didn't have any powers at all, observe:

```
mopsy = Pet.new
```

```
mopsy.super_powers
  => nil
```

Poor mopsy. We can remedy this situation by giving mopsy a basic superpower when she is initialised. Lets do this now.

```
class Pet
  def initialize(args = {})
    @super_powers = args[:power] || "Ability to hop really really quickly"
  end
end
```

Now when we recreate mopsy, she comes already tooled up

```
mopsy = Pet.new
mopsy.super_powers
  => "Ability to hop really really quickly"
```

We can also do this:

```
mopsy = Pet.new :power => "none worth mentioning"
mopsy.super_powers
  => "none worth mentioning"
```

If you need to see a list of all mopsy's attributes you can do so using inspect like so:

```
mopsy.inspect
  => "#<Pet:0x102f87fe8 @super_powers=\"Ability to hop really really quickly\">"
```

Ruby Objects are Open

Because Ruby is an interpreted language objects are open and can be modified at runtime. Classes can be reopened at any time.

We can give mopsy new methods, even after she has already been created. Observe:

```
class Pet
  def shoot_fire
    puts "now shooting fire"
  end
end

mospy.shoot_fire
  => now shooting fire
```

Mopsy can still play chess. The Pet class was added to, not overwritten

```
mopsy.play_chess
  => Now playing chess
```

Inheritance

Rails supports single object inheritance. This means a class can have one parent class and inherits all that class' methods and attributes. We define inheritance relationships using the < operator.

For example, let's say we'd like to define a particular type of pet, say a small and fluffy kitten. Let's create a kitten class that can inherit from our Pet class:

```
class Kitten < Pet
  def play_tennis
    puts "I am now playing tennis"
  end
end
```

Our kitten now has all the attributes of a pet. It can shoot fire from it’s eyes, and play some good chess, but in addition it can also play tennis:

```
tiger = Kitten.new
tiger.play_tennis
  => I am now playing tennis
tiger.shoot_fire
  => now shooting fire
```

Operator Overloading

Did I mention that in Ruby everything is an object? This extends to operators, such as +, -, * and /. Operators in Ruby are actually methods, and we can define and redefine them, like so:

```
class Pet
  def +(pet)
    p = Pet.new
    p.super_powers = self.super_powers + " and also " + pet.super_powers
    return p
  end
end
```

Here we have defined a plus method that receives another pet. This simply creates a new pet with the combined superpowers of it’s two parents and returns it. Observe the offspring of Mopsy and Flopsy:

```
cottontail = mopsy + flopsy
cottontail.super_powers
  => "Ability to hop really really quickly and also Flight"
```

Modifying an Existing Class

As we’ve mentioned before existing classes can be extended. This includes built in Ruby classes. This is a feature that can be used both for good, and for evil:

Good example

```
class String
  def put_times(n)
    for i in (1..n)
      puts self
    end
  end
end

"Marmalade Toast".put_times 5
```

Evil Example

```
class Fixnum
  def *(num)
```

```
    self + num
  end
end

puts 5*4
  => 9
```

Yes, Ruby lets you do this. Be careful and do things and your code will read like liquid sunlight.

Monkey Patching

Reopening code in this way is often known as monkey patching. We can modify or extend any existing class at runtime, even built in classes like strings and arrays. This can be used to great effect, for example Rails Fixnum date extensions, which allow you to type things like:

```
Date.today + 5.days
```

Here Fixnum has been monkey patched with an function that allows it to work with dates and times. This is a nice syntax, although it makes some people cross as it appears to break encapsulation.

Monkey patching is fun, but use it with care, otherwise you'll end up with a twisted mess on the floor.

Object Exercises

Everything is an object, so it's important that we get a whole lot of practice in with making them. In this series of exercises we will create a class, implement some methods, overload some operators and create some virtual attributes.

Define your class

1. Define a class either for a lethal warship or a fluffy animal, depending on your mood today.
2. Add methods so you can set a name for your gunship/fluffster.
3. Add methods to get and set an age in milliseconds.
4. Write a method that retrieves the age in days. You should use the same age attribute as in 3, just modify it before returning it.
5. Write a method that sets the age in days. Again, use the same age attribute, just monkey with it a bit before setting it.
6. Write a getter that returns the age in weeks. (age in days / 7)
7. Write getters and setters that return a string representing a standard action, e.g. “wriggle cutely”, or “blow things up with extravagant firepower”
8. Implement the + operator. Have it return a new object with the names and standard actions concatenated.
9. Implement the * operator. Have it return an array containing multiple instances of the object. (either copies, or several variables pointing to the same object, free choice.)

Receiving a hash

Write an initialiser method that accepts a hash and sets default values appropriately. You should be able to call it like this:

```
Fluffster.new :age => 2, :name => "Floppy"
```

Read only virtual attributes

1. Add x and y position attributes, with getters, but no setters. Initialise the position attributes to be zero.
2. Add move north, south, east and west methods. When these methods are called they should modify the x and y positions.
3. Add a distance_from_home method that computes the current distance from home.
4. Add an at_home? method that returns true if the person is at home, i.e. if their x and y coordinates are zero.

Sorting your custom class

1. Implement the spaceship operator on your class. The spaceship operator returns either –1, 0 or 1 depending on whether the first operand is less than, the same as, or greater than the second one.
2. Now create an array of objects based on your class, and sort them.
3. Pass Array.sort a block, and sort the array based on different criteria.

Monkey Patching Exercises

In this exercise we're going to extend some existing classes.

1. Extend the Array class with a method that iterates over every other element. It should receive a block, and should apply that block to every even element in the array. Call it like this:

   names.each_even {|name| puts name}

2. Extend the FixNum class with a green_bottles class that returns the lyrics for the popular song. For bonus points, make it accept a block that receives the song line by line.

Modules (Mixins)

Modules can be used to add reusable functionality to a class. They are sometimes known as Mixins. A module consists of a whole bunch of methods. By importing it into a class, we gain access to all those methods. This is a handy way to get around the restrictions of single object inheritance, since we may import as many modules as we like.

Defining Shared Functionality

Lets teach flopsy how to make an omelette. Then she will be able to help out in the kitchen. It would be nice if our omelette could accept a few options, so lets allow that too.

```
module CookOmelette
  def cook_omelette(args={})
    number_of_eggs = args[:number_of_eggs] || args[:eggs] || 2
    cheese = args[:cheese] ? "cheese" : nil
    ham = args[:ham] ? "ham" : nil
    mushrooms = args[:mushrooms] ? "mushrooms" : nil
    ingredients = [cheese,ham,mushrooms].delete_if{ |ingredient|
ingredient.nil? }
    ingredients = ingredients.join(" & ")
    "#{ ingredients } omelette with #{number_of_eggs} eggs".strip
  end
end
```

Now include the mixin in the Pet class.

```
class Pet
  include CookOmelette
end
```

All our pets can now make delicious omelettes. Observe:

```
mopsy.cook_omelette
  => "omelette with 2 eggs"
mopsy.cook_omelette :ham => true, :cheese => true, :eggs => 4
  => "cheese & ham omelette with 4 eggs"
```

Module Inheritance

Including a mixin in a class adds those methods to that class as though they had been defined within that class.

Methods added to a class by a module are inherited by subclasses of that class. For example, by including the CookOmelette mixin in the Pet class the Kitten subclass and all its instances also gain that method.

Module Exercises

Modules are a way to extract common functionality into a separate file that can be included elsewhere. Let’s define a module now.

Extracting Common Functionality

This exercise extends the lethal warship of fluffy kitten class .

1. Lets Create a module now. Extract the age and age methods into a module that can be included elsewhere. Name the module sensibly. Now remove these from your class, and instead import the module. You now have the ability to make anything have an age, and to query it’s age sensibly.
2. Write a stereo module that allows your class to play some cool random sounds (really strings). Add it to your class.

Blocks

Blocks are where the fun really starts. A block in Ruby is a sort of unnamed function that can be passed to a method using a special syntax. It can be as long or as complicated as we like, and can span many lines. Here’s a simple example:

```
5.times {puts "Ruby Ruby"}
```

```
Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
```

The block here is the bit of code: puts “Ruby Ruby”. This piece of code is passed to the times method. We use blocks for everything in Ruby, most notably when looping, but in plenty of other ways too. Come with me now as we enter the world of loops…

Why is this cool?

This is super great because it means that the number 5 knows in itself how to iterate up to itself. We have perfect encapsulation. We don’t need a for loop. We don’t need to make any assumptions about implementation. It’s all hidden under the surface.

Passing parameters to a block

We can have our function pass parameters to it’s block. We do this using the | | “chute” syntax. For example:

```
5.times{|i| puts i}

0
1
2
3
4
```

Here the times method passes a number to the block, which is available in the variable i. This is one of the many ways in which we do iteration in Ruby.

Blocks are better than loops

One of the most common and programmer friendly applications of blocks is in looping. Many objects, most notably, Arrays and Hashes will accept a block then apply that block to each of their members in turn. All the looping code is encapsulated, meaning we don’t need to worry about the internal structure of the array.

```
people = ["jim","harry","terrence","martha"]
people.each { |person| puts person }

  => "jim"
  "harry"
  "terrence"
  "martha"
```

As we saw earlier, the Fixnum.times method works in a similar way:

```
5.times { puts "Ruby" }

  => Ruby
  Ruby
  Ruby
  Ruby
  Ruby
```

If we wanted to iterate over some specific numbers, we might use the upto method to create an Enumerable object, and then iterate over that. Observe:

```
5.upto(10) {|i| puts i}

  => 5
  6
  7
  8
  9
  10
```

We could also iterate over a Range object. more on Ranges shortly:

```
(6..8).each {|i| puts i}
  => 6
  7
  8
```

The most common use of a loop, iterating over an array is totally covered, and in fact, when writing Ruby code, we almost never write the sort of looping constructs you might be used to. Blocks have them covered

each_with_index

If for some reason we need t get the index of an array, we can do this too using Array.each_with_index.

This method accepts a block with two parameters, the value and an index. We can use it like so:

```
people = ["jim","harry","terrence","martha"]
people.each_with_index { |person, i|   puts "person: #{i} is called #{person}" }
  => person: 0 is called jim
  person: 1 is called harry
  person: 2 is called terrence
  person: 3 is called martha
```

Short and Curlies

We can declare a block in two ways. We can use the curly braces syntax, or the do/end syntax.

The difference between them is that the curly braces syntax can only take a single line of code, whereas the do/end syntax can take as much code as you like, even an entire web page template if needed.

Here is an example of the curly braces syntax:

```
favourable_pets = ["kittens","puppies","hamsters"]

favourable_pets.each_with_index { |pet, i| puts pet }
  => kittens
  puppies
  hamsters
```

And here is an example of the do/end syntax. Notice the code is more spread out. This is more readable for large blocks of code.

```
favourable_pets.each_with_index do |pet, i|
  puts pet
  puts i
end

  => kittens
  0
  puppies
  1
  hamsters
  2
```

Sorting an Array Using a Block

By default array.sort will sort any array of values using the <=> spaceship method of the value. This method as we have seen returns –1, 0 or 1 depending on whether the value is lower, equivalent or greater than another value. The String class implements the <=> method by comparing the two values alphabetically, so by default array.sort sorts an array of strings alphabetically like so:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort
  => Derek
  James
  Stuart
  Thomas
```

If we want to override this, tehre are two easy ways to do it. First we can override the spaceship operator (coming soon). Alternately, we can pass Array.sort a block which can be used instead. For example the following code sorts an array in order of the second letter:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort {|a,b| a[1] <=> b[1]}
  => James
  Derek
  Thomas
  Stuart
```

Nice and Simple.

Modifying each element of an array with map

Map is an insanely useful array function that lets you modify each element of an array using a block.

Say you have an array of strings:

```
['kittens', 'puppies', 'hamsters']
```

Lets say you want to convert this to an array of symbols, you could do something like

```
['kittens', 'puppies', 'hamsters'].map {|i| i.to_sym}
```

You will now get

Writing a method to accept a block with yield

So how does this actually work? How does a method yield control to a block? Well, we use a keyword called yield. For example, the following (very simple and not terribly useful) function accepts a block of code and then simply runs it once:

```
def do_once
  yield
end

do_once {puts "hello"}
  => "hello"
do_once {10+10}
  => 20
```

You can call yield as many times as you like. This is how the Array.each method works, by iterating over the array and calling yield for each item, passing the item as a parameter. Here's a silly example:

```
def do_thrice
  yield
  yield
  yield
end

do_thrice {puts "hello"}
  => "hello hello hello"

do_thrice {puts "hello".upcase}
  => "HELLO HELLO HELLO"
```

Passing Parameters to a Block

A block is an unnamed function and you can easily pass parameters to it. The following example accepts an array of names and for each one sends a greeting to the block. The block in this case just puts the greeting to the screen.

```
def greet(names)
  for name in names
    yield("Hi There #{name}!")
  end
end

greet(["suzie","james","martha"]) { |greeting| puts greeting }
  => Hi There suzie!
  Hi There james!
  Hi There martha!
```

What to Do Now?

Congratulations! Well done Ruby developer. Now attempt the following exercises on blocks and yield. Marks are awarded for clarity of thought, and beauty of implementation.

Blocks Exercises

The following exercises are designed to give you experience working with blocks. Block can seem a little weird and backwards to start with, but once you get the hang of them you'll wonder how you lived without them.

Counting with Blocks Exercise

Blocks are unnamed functions that can be passed to a method. They are used a lot in Ruby so it's important that we get good at them. Lets have a go now…

1. Using the Fixnum.times method, write code which prints "Hello Ruby" to the screen 100 times.
2. Modify your code so that it prints the numbers 1 to 100 to the screen.

Sorting arrays with Blocks Exercise

Array.sort predictably allows us to sort an array. It accepts a block with two arguments, like so:

```
[1,4,2,3].sort {|a,b| b <=> a}
```

This code sorts the array in reverse order. Note that the sort method creates and returns a new array. It is non-destructive, since the original array still exists unchanged. If you want instead to change the array in place you would use:

```
[1,2,3,4].sort!
```

Now attempt the following exercises:

1. Create an array of strings. Use the Array.each method to iterate over the whole array, printing out the strings to the screen.
2. Use the Array.sort method to sort the array.
3. Use Array.sort to sort the strings by length, rather than alphabetically. Do this by passing Array.sort a block which returns –1, 0, or 1 depending on whether the first string is shorter, the same length as, or longer than the second.

Extra hard secret bonus questions:

1. Create an array of integers. Sort it so that the numbers, when modded by 10, are in order, so for example:

   [1, 120, 19, 13, 906]

would become:

```
[120, 1, 13, 906, 19]
```

since 120 % 10 is 0.

1. Create an array of integers, 1 to 52. Now shuffle them using Array.sort. How will you do this?

Iterating over an array Exercise

We can pass a block to an array to be executed on each item in that array. For example:

```
['fork','knife','spoon'].each {|table_item| puts table_item}
```

Our code doesn't need to know the internal details of the array, it just passes a block, and lets the array sort itself out.

1. here is an array of strings. Pass a block to Array.each printing them out one by one.

   [‘cats’,’hats‘,’mats’,’caveats’]

2. Modify your code to only print out only strings which contain the letter ‘c’
3. The each_with_index method accepts a block with two parameters, the value, and the index. Use it to print out the strings in the array preceded by their index in the array like this:
   1. cats
   2. hats
   3. mats
   4. caveats
4. Repeat the exercise above, but now only print out every string with an odd number index.

Text Replacement with blocks and and gsub Exercise

The String.gsub method will find and replace substrings in text. It’s terribly useful, but sometimes we need more, we need to find and manipulate strings in text.

Say you have a string containing URLs, maybe culled from twitter. You could replace all the urls like this:

```
tweet_text = "hello http://www.google.com hi"

tweet_text.gsub(/http:\/\/[^ ]*/, "A URL was here.")

  => "hello A URL was here. hi"
```

This is OK, but what if we wanted to replace the URL with a functioning link. The gsub method will optionally accept a block. The block receives a parameter which contains the match. The block must then return a value which is used to replace the match.

1. Create a string containing one or more URLs. Now write code using gsub to hyperlink all the urls.
2. Assume your string is a tweet. Now write code to hyperlink all the hash tags. Hashtags start with a # and end with a space or newline.
3. Finally write code to hyperlink all the user names. User names start with @.

Captialise each element

We can create an array of strings like this:

```
%w(kittens puppies gerbils hamsters)
```

This is shorthand for:

```
['kittens', 'puppies', 'gerbils', 'hamsters']
```

Use map to convert this array to this:

```
['Kittens', 'Puppies', 'Gerbils', 'Hamsters']
```

Convert to underscores

Say we have an array of Java style method names camel case method names like this:

```
['goNorth', 'goSouth', 'steadyAsSheGoes']
```

We want to convert these to proper Ruby snake case method names, like this:

```
['go_north', 'go_south', 'steady_as_she_goes']
```

More with arrays Exercise

Array.delete_if accepts a block. It removes elements from an array if they match certain criteria.

1. Make an array of strings. Use Array.delete_if to remove any strings that are over a certain length

Print Times Exercise

1. Write a method that accepts an integer and a block and runs the block the number of times specified by the integer. You should be able to call it like this.

   number_of_times(10) {| i | print “this is time #{i}”}

Answers

Counting With Blocks

1. We can do this simply using 10.times and passing a block

   10.times {puts “ruby”}

2. The times method can pass a variable to the block.

   10.times {|i| puts i}

Sorting Arrays with Blocks

```
a = ["ant","herbivore","cheshire"]
a.each {|str| puts str}


a.sort

a.sort {|a,b| a.length <=> b.length}
```

Extra hard secret bonus questions:

```
a.sort{|a,b| a%10 <=> b%10}
```

1. You could create this array by typing out all the numbers, or even by using a for loop, but the simplest way to create an array of integers is to start with a range, and convert it using to_a. We will cover ranges shortly

   cards = (1..52).to_a

We can then shuffle the deck by sorting randomly, eg:

```
cards.sort {rand <=> rand}
```

You could also have:

```
cards.sort {rand - 0.5}
```

This works because rand - 0.5 will produce a negative or a positive number at random.

Iterating Over an Array

1. The simplest way is to pass a block to array.each, which receives each item and puts it to the screen.

   [‘cats’,’hats‘,’mats’,’caveats’].each {|item| puts item}
2. A simple if statement in the block will only output strings that include the letter c.

   [‘cats’,’hats‘,’mats’,’caveats’].each {|item| puts item if item.include? ‘c’}
3. each_with_index passes two values into the block. Then we compose them into a string and put them on the screen.

   [‘cats’,’hats‘,’mats’,’caveats’].each_with_index { |item,i| puts “#{i}. #{item}” }
4. Another if statement, this time on i. If i is odd, put the item on the screen.

   [‘cats’,’hats‘,’mats’,’caveats’].each_with_index { |str,i| puts “#{i} #{str}” if i.odd? }

You could also split this over 7 lines to make it easier to read:

```
items = ['cats','hats','mats','caveats']
items.each_with_index do |item,i|
  if i.odd?
    puts "#{i} #{item}"
  end
end
```

Many people prefer the longer format.

Eigenclasses (Static Methods)

In Ruby, all methods exist within a class. When you create an object, the methods for that object exist within it’s class. Methods can be public, private or protected, but there is no concept of a **static method**. Instead, we have singleton classes, commonly referred to as eigenclasses.

No static methods

Static methods are class methods. they belong to the class, not the instance. This would break Ruby’s simple object structure, since classes are instances of class Class, adding methods to Class, would make them available everywhere, which is not what we want.

Singletons

Instead, Ruby lets us define an unnamed singleton class that sits in the inheritance tree directly above any object. Lets do this now and create a static method.

```
class Kitten
```

```
  class << Kitten
    def max_size
      8
    end
  end
end
```

The class << Kitten syntax opens up the eigenclass and pops the max_size method within it. We can then access it like this

```
puts Kitten.max_size
```

Notice that we are talking to the Kitten class as an object here.

Shorthand

We use the class << self syntax to explicitly open an object's eigenclass. We can accomplish the same thing using the shorthand syntax:

```
def Kitten.max_size
  8
end
```

This adds a method to the Kitten eigenclass. We can also add a method to the eigenclass of any other object, like so:

```
fluffy = Kitten.new
popsy = Kitten.new

def popsy.deploy_wheels
  @wheels = :deployed
end

def popsy.launch_scouter
  @scouter = :launched
end
```

Here we have added a method to popsy's eigenclass, allowing her to deploy wheels.

Eigenclasses in the Inheritance Hierarchy

The eigenclass sits directly above the object in the inheritance hierarcy, below the class of the object. It provides a handy place to put methods hat we want to apply directly to the object, rather than to every instance of that object. It feels technical, but once you get it, it's actually rather nice.

Adding a method directly to an object

In irb (or in a ruby file) create 3 instances of your warship/pet class. Add a different method to each of them. Verify that only the instance you added the method to it to can call it.

You're writing to an eigenclass. Feels natural doesn't it?

Exception Handling

Exception handling in Ruby is very similar to other languages.

Raising an Exception

Raising an exception in Ruby is trivially easy. We use raise.

```
raise "A Error Occurred"
```

This will raise the default RuntimeException.

Raising a Specific Exception

We can also raise a specific type of exception:

```
value = "Hi there"
raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
```

Recuing Exceptions

We can rescue exceptions easily. Put the code that might raise an exception in a begin, rescue end block. If an exception occurs, control will be passed to the rescue section.

```
begin
  raise "A problem occurred"
rescue => e
  puts "Something bad happened"
  puts e.message
end
```

Rescuing Specific Exceptions

We can rescue different types of exceptions

```
value = "Hi there"

begin
  raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
  raise "A problem occurred"
rescue TypeError => e
  puts "A Type Error Occurred"
  puts e.message
rescue => e
  puts "an unspecified error occurred"
end
```

The Ruby Exception Hierarchy

Here are the built in exceptions available in Ruby:

```
Exception
 NoMemoryError
 ScriptError
   LoadError
   NotImplementedError
   SyntaxError
 SignalException
   Interrupt
 StandardError
   ArgumentError
   IOError
```

```
      EOFError
    IndexError
    LocalJumpError
    NameError
      NoMethodError
    RangeError
      FloatDomainError
    RegexpError
    RuntimeError
    SecurityError
    SystemCallError
    SystemStackError
    ThreadError
    TypeError
    ZeroDivisionError
  SystemExit
  fatal
```

Defining Your Own Exception

You can define your own exceptions like so:

```
class MyNewError < StandardError
end
```

You can then raise your new exception as you see fit.

Exception Exercises

Try these exercises to get a feel for exception handling in Ruby.

Raising an Argument Error

Extend your kitten class from yesterday. Lets assume your kitten needs an age (0 will not do) Raise an argument error if age is not set in the initialiser

Raise a Type Error

Your kittens age must be a Fixnum. Check for this, if it is not, throw a Type Error

Catching a Division By Zero Error

Can your kitten do maths? If not, write a divide function now that accepts two values and divides them. Catch the division by zero error, and if it occurs, return nil.

Ruby Challenge Exercises

If you're ahead of the groups, tackle the following Ruby problems in whatever way you see fit. Points will be awarded for conciseness, readability and extreme cleverness.

Writing Concise Ruby Exercise

1. Write a single line of Ruby that swaps the contents of two variables.
2. Write a single line of Ruby that gets the last element from an array. Hint, there are

several ways to do this.
3. Write a single line of Ruby that reverses an array.
4. Write a single line of Ruby that converts an array into an html unordered list.
5. Write a single line of Ruby that reverses all the words that start with the letter a in a string.

Hashes Exercise

1. Create a hash to hold a list of feelings and foods. It should look like something like this, only longer:

   food_hash = { :happy => “icecream”, :pensive => “witchetty grub” }

2. Write a program that allows a user to type in a feeling and have it return the corresponding food (Tip: use to_sym to convert the user’s input into a symbol)

Advanced Ruby

Meta Programming

Metaprogramming means writing Ruby which writes Ruby. We can automate the generation of code.

Monkeypatching

The simplest way to extend Ruby is to simply open up a class and add methods on it.

Here we add a handy summarise method to String, which lets you get the first few characters, always breaking at a word boundary.

```
class String
  def summarize(l=200)
    i = 0
    self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
  end
  def summary_is_full_text?(length=200)
    return self.summarize(length) == self
  end
end
```

Opening up classes like this is known as monkeypatching.

send

Respond to lets us check if an object responds to a method.

The send method allows us to call a method using a string.

On my site, widgets are little bits of content that can be placed on a page. Say we have a Widget class. There are lots of different types of widget, and editing them requires slightly different setup.

So I have an new method on my controller that makes a widget and sets up some values to let me display a form:

```
def new
  @type =  params[:widget_type]
  @widget = Widget.new
  @widget.widget_type = @type
  if @type == "pie_widget"
    @products = Product.all
  elsif @type == "news_widget"
    @news_list = News.order("date DESC")
  end
end
```

This has a big old untidy if else statement in it, which will only get bigger as I add extra widget types. Ideally I’d like to break these out into separate methods

```
def new
  @type = params[:widget_type]
  @widget = Widget.new params[:widget]
  @widget.widget_type = @type
  send(:"new_#{@type}")
end

protected

def new_pie_widget
  @products = Product.all
end

def new_news_widget
  @news_list = News.order("date DESC")
end
```

I now configure each type of widget in its own method which is more readable.

respond_to?

What if I add a type of widget which doesn’t require configuration, say a plain text widget. I might write this:

```
def new
  @type = params[:widget_type]
  @widget = Widget.new params[:widget]
  @widget.widget_type = @type
  send(:"new_#{@type}")
end

protected

def new_pie_widget
  @products = Product.all
end

def new_news_widget
  @news_list = News.order("date DESC")
end

def new_text_widget
end
```

The empty method is a bit lame though, lets sort that out:

```
def new
  @type = params[:widget_type]
  @widget = Widget.new params[:widget]
  @widget.widget_type = @type
  if respond_to?(:"new_#{@type}")
    send(:"new_#{@type}")
  end
end
```

Now we check to see if the object can respond to the method before we try to call it. If there is no new_text_widget method, we just ignore the send and move on.

define_method

We can also define methods on the fly. This is useful if we need to define several methods which are similar to each other.

```
%w(product news recipe) do | name |
  define_method "#{name}_id=" do |id|
    self.model_id = id
    self.model_name = _name
  end
end
```

Here we define three methods: product_id=, news_id= and recipe_id=

Each of these sets the value for model_id and model_name correctly.

Introduction to Rails

Hallo and Welcome to this intensely practical, wildly exciting, whirlwind tour of the Ruby on Rails web development framework. Rails is an intensely pragmatic , developer centric Rapid Application Development Framework that gets people all of a lather, since it allows a lone developer, or small team to create pretty much anything imaginable in double quick time.

It’s favoured by startups, since it allows you to try out bunch of ideas lickety split. In fact it’s almost as fast to write Rails as it is to write specifications. This enables iterative design, where we build a little code, show it to the client, and change it according to their feedback. This means happy clients and engaged, super productive developers.

So, without further adieu, lets get this party started.

Hello World

Hello World is the traditional way to start any new framework. Hello World in Rails is fairly simple, around six lines of code. In this section we’ll create an MVC hello world using a simple model, a controller, and a view, in fact the full MVC stack.

Create your rails application

At a terminal type:

```
rails new HelloWorld
```

This initialises a new blank application called HelloWorld

Start your rails application server

At a terminal navigate to your HelloWorld directory and type:

```
ruby script/rails server
```

Wait for the application to start then in a web browser visit:

```
http://localhost:3000
```

Creating a Controller

Create a new file in the app/controllers directory called hello_controller.rb

```
class HelloController < ApplicationController
  def index
    @text = "Hello World!"
  end
end
```

This controller defines a single method called index. Index is the default controller action. If you’re familliar with HTML this should make sense.

The index method creates an instance variable called @text.

Creating a View

In the app/views/hello directory (you'll need to create this directory) create a file called index.html.erb containing the following

```
<h1>Hello World</h1>
<%= @text %>
```

Views have access to the controller's instance variables. You don't need to do any work to pass them through, they're just there.

Create a route

Visit config/routes.rb. Take a moment to read the helpful instructions, then add the line:

```
match '/hello' => 'hello#index', :as => :hello_world
```

This line creates a named route (called hello_world). It matches the

Hey Presto

In a web browser visit:

http://localhost:3000/hello

And there we are. Of course we're only touching on the awesome power of Rails here, but this should give you an idea of how easy things are to get going.

Hello World Exercises

Let's play with Hello World a little

Change the text

Adjust your app so that instead of printing hello world, it prints a different string.

Change the URL

Adjust your app so it responds to the root url. You should be able to visit:

```
http://localhost:3000/
```

and see hello world.

- First delete public/index.html
- Now view the instructions in config/routes.rb to create a default URL.

Refactor

Change your app into a goodbye world app. Here are the thing's you'll need to do:

- Add a new controller goodbye_controller.rb.
- Add a route.
- Add a view

Some Theory

It’s pretty cool to get something working, and it’s always impressive to do something quickly, but without a theoretical underpinning we won’t get far.

We’ve built a little hut here. How lets go back and add the foundations.

So how do you build websites?

All websites are the same: HTML, plus CSS, maybe a few images, and possibly a few other assets, Flash, JavaScript and the like. The user sends a request by hitting a URL with their browser. Sometimes the request is made automatically by JavaScript, and we call this AJAX. Sometimes the user hits the site using another app, and we call it a web service or API, but the principle is the same. A request is made and data is shipped back.

What happens between the request and the return is a different matter. Maybe you have HTML files sitting and waiting to be shipped out. Maybe you have a template that fills itself in from a database. Maybe you have a whole other server that spits out HTML.

A naive approach to web design

One way to tackle this is to write a single file which queries the database, pulling any data that’s required, then writes it directly to an HTML file, adding any headers and footers that might be needed. This if OK for a simple site, it’s how many PHP and classic ASP websites work, but what happens if we start to get more complicated.

How Rails handles a request.

Rails uses a pattern called MVC, but you could call it “a place for everything, and everything in it’s place”. It’s a general purpose design pattern which encourages separation of responsibilities, the view from the request logic from the business logic.

The Router

The first thing that happens when a Rails instance receives a request is it gets passed to the router. The router inspects the URL and request header, passes it through a series of rules written by you, and decides which method on which controller to pass it to.

The Controller

Assuming the router manages to interpret the URL correctly, it passes the request to a controller. A controller is just a Ruby object that has various methods for handling requests. Any URL or post parameters are separated out into the params hash automatically and become available.

Pulling models

The controller can now pull up any models that are required. A model is a simple Ruby object that represents something. For example, if you are writing a blog, the model might be a blog post. If you are writing a social network, the model might be a person. You might have an array of models. Models can be taken from the database, can be generated right there in the controller, or can be made in some other way.

Calling the View

Having pulled or made any models, the controller now passes them to a view. A view inspects the models and generates some output which can be sent to the user. Most of the time this is HTML, but you can also output XML, PDF, XLS, JSON or any other format, depending on the request.

Helping the view

The view has access to helper methods which are written in plain Ruby. These are defined in helper classes. You can split out tricky logic into helpers, keeping your views clean.

Rails Idioms: That's some RESTful DRY CRUD you've written

Here are some ideas that Rails people hold dear to their hearts.

MVC

Model View Controller. Put your code in the right place.

The Model contains your business logic, separated into simple, easy to understand methods.

The View is responsible for writing a file that can be sent to the user. It contains as little code as possible. It calls methods on the model to get any data it needs.

The Controller links the two together. It generates the model and passes it to the view, and optionally passes user parameters back to the model and saves it.

DRY: Don't Repeat Yourself.

Don't repeat yourself. Write code once, in the correct place. Repetition leads to bugs.

Fat Model, Skinny controller

The controller should do very little. Ideally it should just make, and call methods on the models. The models implement a series of simple, well defined, easy to understand methods that do most of the actual work. A place for everything and everything in it's place. Clean, easy to see, testable.

CRUD

Rails is a system thats optimised to one thing particularly well: CRUD

- CReate a resource
- Update a resource
- Destroy a resource
- (also Show a resource and Index - perform a default action)

When you think about it most things you will ever want to do in a web application fall into one of these categories.

eg. A user signs up for your application and views there profile page

Your web app Creates a user and then Shows that user eg. An administrator writes a page and then edits it

Your web app Creates a page and then Updates it

eg. A user views the pages in a category

Your web app performs the default Index page action which you have decided will render a list of pages.

eg. A user signs in

Your web app hits the Session controller and calls new.

eg. A user signs out

That would be the Session controller again, and the destroy method.

Crud forms the basis of everything we do in Rails. Work within it and you'll have clean, readable, maintainable code.

When you think about it, most of the things you might want to do on a website fit into CRUD. You may need to break it sometimes, but try not to too often. Why not? A place for everything, and everything in it's place.

REST

REpresentational State Transfer sounds complex but really just refers to the fact that HTML headers carry more information than just the URL. When the user makes a request via a form for instance, that's a POST request. Requests to the same URL, but with different request types can be handled differently by the Rails server.

Creating a simple site with scaffold

Scaffolding allows us to throw together mockups very quickly which can later be fleshed out into real apps. A scaffold creates a CRUD interface for us

This course is practical, so now, we're going to make something that works. We'll come back to the theory afterwards, but for now, lets get our hands dirty.

This exercise will be in four parts.

1. First we'll create a server.
2. Then we'll start it up.
3. We'll hook it up to a database, and check that it works.
4. We'll create a fully featured Rails application using a scaffold generator.

Step 1: Creating a server

To get started, you'll need a command line or terminal. Get it up and type:

```
rails new superfly
```

Wait an moment, now look, it's made some code for you. In a minute we'll take a look at what we've got, but for now, let's just start the server.

Step 2: starting the server

Move into the superfly directory and type:

```
rails server
```

or just

```
rails s
```

If all has gone to plan, the server should start and let you know it's listening on port 3000. Now pull up a web browser, I like Firefox, and visit:

```
http://localhost:3000
```

See a welcome message? That's great! You're now riding the Rails.

Step 3: Hooking up the database

Since we're making a real application, we have a little housework to do. Don't worry, it won't take long. You'll need to choose which type of database you're going to use. Common choices are:

SQLite: If you want an instant, low overhead database that's stored in your filesystem. Perfect for development, but good enough for production too.

MySQL: If you want an arguably more robust database that can also be used on a production server.

Of course if you want to use more than one type of database that's fine too, though maintaining consistency between your development and production environments is encouraged.

What you'll need to hook up the database

You'll need three things to do this

1. A database.yml file to tell rails the database type, name, username and password you want to use. You'll find that a sample database.yml file has been created for you in the config directory.
2. An entry in the Gemfile, which tells Rails which database adapter to use. The default is sqlite3-ruby.
3. The Gem itself. Gems are reusable bits of Ruby code which extend Ruby's functionality. You'll need a Gem that can talk to your database. In Rails 3 and above we manage and install gems using a piece of software called Bundler, which is itself a gem.

Installing the Bundler

Before you begin, you'll need to install the Bundler gem. You'll only need to do this once. At a command line type:

```
sudo gem install bundler
```

You will need to have root privileges to do this. The gem will be pulled from the internet and installed locally.

The bundler lets you bundle gems into your rails application.

Now choose your database

You are standing at a crossroads. Which road will you take? If you plan on using SQLite, skip to the section on hooking up an SQLite database. Likewise for MySQL.

Hooking up an SQLite database

Congratulations. This is the default, so you'll have the least work to do.

Edit database.yml

The default settings should be good here. SQLite databases are created on the fly, on demand, so you shouldn't need to make any changes. Your file should look something like this:

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

test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

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

This file defines 3 databases for the three different modes, or "environments" in which Rails can run. The development and test databases will be used by us while we're working on the project locally. The production database will be used by the live server. We can add more environments if we wish. We might want to add a staging environment for example.

Do please note that spaces matter in yaml. If you get a syntax error, it may be because of a missing space after a colon

Edit the Gemfile

Now have a look at Gemfile. We should see a line like this:

```
gem 'sqlite3-ruby', :require => 'sqlite3'
```

This line tells Rails to use the sqlite gem. We don't need to change this.

Run Bundler

Now we need to run bundler to check our gems so at a command line type

```
bundle install
```

This does two things. It will install any missing gems, and it will write to a file called Gemfile.lock which tells Rails which gems are installed. The gemfile enforces consistency between environments. We'll look at it in more detail later.

Hooking up a MySQL Database

Using MySQL is a good choice, but you'll have a little more work to do as it's not the default.

Edit database.yml

You'll see a short file that defines 3 databases called development, test and production. By default this is set up to use SQLite, so we'll need to make some changes. A typical database.yml config file defines three databases and looks something like this:

```
development:
  adapter: mysql2
  database: superfly_development
  username: root
  password: my_password
  timeout: 5000

test:
  adapter: mysql
  database: superfly_test
  username: root
  password: my_password
  timeout: 5000

production:
  adapter: mysql
  database: superfly_production
  username: live_user
  password: live_password
```

There are three databases defined here, which correspond to the three different modes, or "environments" in which Rails runs by default. The development and test databases will be used by us while we're working on the project locally. The production database will be used by the live server. We can add more environments if we wish. We might want to add a staging environment for example.

Do please note that spaces matter in yaml. If you get a syntax error, it may be because of a missing space after a colon.

Make the database

First you'll need to create one or more databases. You'll at least need a development database, and preferably a test database too. I use a tool called sequel pro for this, but you could use the command line, or another piece of software that lets you talk to a mysql database, for example SQLYog or Sequel Pro.

Edit the Gemfile

Now have a look at Gemfile. We should see a line like this:

```
gem 'sqlite3-ruby', :require => 'sqlite3'
```

This line tells Rails to use the sqlite gem. We want to use the mysql database gem instead, so delete this line and replace it with:

```
gem 'mysql2'
```

This tells Rails we will need access to the mysql database.

Run Bundler

Now we need to run bundler to check our gems so at a command line type

```
bundle install
```

This does two things. It will install any missing gems, and it will write to a file called Gemfile.lock which tells Rails which gems are installed. The Gemfile enforces consistency between environments. We'll look at it in more detail later.

Testing our database configuration

To test our configuration, we will use a piece of software called rake. Rake stands for Ruby Make, and it's a little program that lets us define and run routine little tasks on our Rails server. More on rake later.

Open up a terminal and type:

```
rake db:migrate
```

At this point, either everything will run smoothly and your server will connect, or you will get errors. If you get errors, read them and do what they say.

Housework complete

You should now have a functioning Rails instance that can start a server and connect to a database. Now on the the good stuff.

Step 4: Super super quick start with Scaffold

One of the wow features of Rails is the scaffold generator. It lets you write a whole lot of code very fast. Pros tend not to use it, but it's a good way to demonstrate what can be done in next to no time.

Open up a terminal and type the following:

```
rails generate scaffold SuperHero name:string powers:text description:text
```

This will, in a stroke, generate all the code needed to allow us to create, edit and view superheros. Lets try it out before we explain what we have done.

Again, visit a terminal and type

```
rake db:migrate
```

This will update our database. More on this shortly.

Start your server with

```
rails server
```

your server should start up. Now in a web browser, go and visit:

```
http://localhost:3000/super_heros
```

You'll find a complete website has been made for you. You can add superheroes, edit them, view a list, drill down to individuals, and then delete them. It's all rather slick, and terribly terribly quick to do.

So what do we have here? Investigate your code.

So what do we have here? Investigate your code.

Look in the app directory. Notice the models, views and controllers directories, here is our MVC. Look in the config directory and notice the routes.rb file. Here is our router.

Make a Blog Rails Exercises

It’s traditional, when starting in Rails, to constructing a blog. It’s a bit like making your own lightsabre, it’s a right of passage that every aspiring Rails Jedi must go through. We’ll start out with a scaffold, then flesh it out.

In this exercise we’ll create a blog with comments.

Make your pages using scaffold

Call Rails new to generate a new project. Create your database and hook it up. run bundle install to get all the gems.

Use the rails scaffold generator to create a blog_post model. Blogs should have a title, author, date, intro text and main text. Make these with the generator and verify they work.

The syntax for a rails scaffold generator (Rails 3.0) is as follows:

```
ruby script/rails generate scaffold title:string intro:text
```

etc…

Cerate your comment model using scaffold

Create a comment model using the scaffold generator. Comments should have content, an email and an optional url.

Wire them together

Create a one to many relationship between comments and blog posts. A one to many relationship means that one post has many comments.

To create a one to many relationship, you need to:

- Add a post_id field to the comment model. The easiest way to do this is to edit your migration, and then run rake db:migrate:reset to rerun all migrations.
- Add has_many :comments to your post model
- Add belongs_to :blog_post in your comments model

Create a comment form

Add a form to your blog post show page to let you add comments. The easiest way to do this is using a partial. You’ll need to prepopulate the post_id field.

Show your comments

Show all the comments on the post show page. Since you have put has_many :comments in your post model, you can call @post.comments to get an array of comments. Iterate over this array on the show view.

Fix the index page

Edit the blog index page so it only shows the title and intro text for each blog post. Show a comment count on this page too.

Fix your Routes

Set your default root url so it renders blog_post#index.

Create a Layout

Edit your layout file. Add a header and footer. Edit the stylesheet import so it imports a different stylesheet. Make your blog look pretty.

Add Validation

1. Edit your BlogPost class to add validation. Use validates_presence_of to test for the presence of the title, intro and content fields.
2. Ensure comments are over 5 characters, and fewer than 500 characters. Ensure the comment and email fields are mandatory.

Users and Sessions

Add sessions. You'll need to run the sessions migration. Look up how to do this.

Create a Person Model

Use the scaffold generator to create a person model. People should have a name, email address, and password. Now we have users.

Create a Session Controller

Create a sessions controller. It should have 3 methods new, create and destroy. Create a view for new which renders a login form. Look at your other forms to see how to do this. On submit, it should direct users to the session create action. Create appropriate routes for your actions. Look in the routes.rb file for instructions on how to do this.

In create, inspect the params hash to see if a user exists with the email and password entered. If they do, add their id to the sessions hash.

You can find a user given an email and password using something like the following:

```
@person = Person.where :email => params[:email], :password => params[:password]
```

The session is implemented as a hash. If a person is found, add their ID to the session hash.

```
session[:person_id] = @person.id
```

Create find_user and logged_in?

Edit your application_controller. Application controller is the controller superclass. Any methods you add here will be available to every controller. Create a method called find_user. This should inspect the session hash, and create a @person instance variable if there is a person_id in the session.

You can do something like:

```
Person.find session[:person_id]
```

Edit your application controller. Add a before_filter that runs the find_user function always. We will now have a person available to us with every request, or nil if there is no one logged in.

Edit your application_controller. Create a logged_in? method. If logged_in is falsey (nil), this should redirect to the sessions#new method.

Lock Down Your Post Controller

Edit your blog_post controller. Add a before_filter that calls logged_in? for every method except index and show. If logged_in returns false, redirect the user to the sessions#new method and return false. This will prevent any more filters from running.

You should now have a blog that you can log into, add blog posts to, and comment on.

More Filters

Add before filters on the comments controller. to prevent people from deleting comments unless they are logged in.

Edit the application_helper. Add another method: logged_in? that returns true if there is an @person instance variable. Helper methods are available in views.

Edit your view so that people cannot see links that they should not be able to see. Something like:

```
<% if logged in? %>
  link
<% end %>
```

Debugging

Debugging Rails

Rails comes with a couple of fully featured debugging systems.

The Console

The rails console lets you fire up an instance of your rails server on the command line. You have full access to Active Record and all of your models. It's terribly good.

Fire it up with:

```
rails console
```

or just

```
rails c
```

You can create models, define methods, and do anything you can do in a regular rails instance.

If you make changes to your code, you can load them in to your console by typing:

```
reload!
```

The Debugger

If you need to poke around inside a running Rails instance to see what's going on, you can use the debugger. To use it you must start your Rails instance in debug mode like so:

```
rails server --debugger
```

or just

```
rails s --debugger
```

Then insert breakpoints into your code by typing:

```
debugger
```

You can put breakpoints in your models, views or controllers. Within the debugger you have access to all of the variables defined at or before the breakpoint. You can change variables, execute code, write ruby, etc, etc.

To exit the debugger, continue with

```
c
```

Tip: Pressing enter will repeat the last console command. If you have multiple debug statements, you can just hold down enter to skip through them.

Debugging Exercise

In this exercise we'll have a go at using the debugging tools.

Debugging Hello World

First, let's use the debugger to debug our Hello World server. Fire up your hello world instance in debug mode. Now insert a breakpoint in the hello controller (or goodbye controller), right after you define the @text instance variable.

Hit the URL, and the app will freeze. Go to your terminal, and you'll find it's waiting for input. You can view the contents of the @text variable, and even change it's value.

Inspecting Superfly

Now let's have a look at Superfly. Navigate to your superfly directory and fire up a console.

```
ruby script/rails console
```

Now try creating some superheros. You can do this by typing things like:

```
hero = SuperHero.new :name => "hatman", :powers => "pouncing, looking chipper"
hero.save
```

Inspect your database, you will find you have actually created a table row.

ActionDispatch Routing

Routing

The router is the thing that takes a url, parses out any variables, and calls the correct method on the correct controller.

Checking your routes

We can check our routes using:

```
rake:routes
```

This lists all the routes currently defined by our system. Commonly we use this in conjunction with grep, so we might type:

```
rake routes | grep hello
```

To find all our hello routes, listed below:

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

Simple Routes

This simplest type of route simply matches a url to a path to a method on a controller. Let’s try this now:

```
match "/hello" => "hello#index"
```

This simply matches the path /hello to the index method on the hello controller. Nice and simple.

We can also pass one or more parameters to our controller from our path like so:

```
match "/say_hi/:greeting" => "hello#index"
```

In out controller and view we will find a params hash which contains the key :greeting. We can access it like so:

```
params[:greeting]
```

This will contain the value that was passed in.

Named Routes

Named routes give us access to some useful shorthand methods in our controllers and views. We can name a route anything we like, like so:

```
match "/hello" => "hello#index", :as => "hello"
```

Naming a route patches our controllers and views with some additional methods. The above creates the following methods:

```
hello_path
hello_url
```

These output urls as strings, the first outputs just the path, the second, the fully qualified URL.

Note that these methods are not available inside your model. In most cases, your model should not know about the URL structure of your website, though there are exceptions.

Restful Routes

You can define which HTTP method your route will respond to. This means you can have several actions listening to the same URL. The one which gets called will depend on whether you are receiving a “post” form submission, or just a “get” request for data.

We do this using via:

```
match "/hello" => "hello#index", :as => "hello", :via => "get"
match "/hello" => "hello#create", :as => "hello", :via => "post"
```

The same url, different methods.

Resource based routes

Rails is really really good at CRUD. Wouldn’t it be great if we could create all our crud routes all in one go. Luckily we can using resources.

```
resources :page
```

Will create CRUD based RESTful routes for a page object.

```
rake routes | grep page

    page_index GET    /page(.:format)           {:action=>"index",
:controller=>"page"}
           POST   /page(.:format)           {:action=>"create",
:controller=>"page"}
  new_page GET    /page/new(.:format)       {:action=>"new",
:controller=>"page"}
 edit_page GET    /page/:id/edit(.:format)  {:action=>"edit",
:controller=>"page"}
      page GET    /page/:id(.:format)       {:action=>"show",
:controller=>"page"}
           PUT    /page/:id(.:format)       {:action=>"update",
:controller=>"page"}
           DELETE /page/:id(.:format)       {:action=>"destroy",
:controller=>"page"}
```

Customising resource based routes.

- Resources accepts a block.
- Create member routes to operate on a single model.
- Create collection routes to operate on all models.

  resources :competition, :only => [:index] do member do post :enter get :congrats_you_won, :sorry_you_have_lost, :already_entered end end

Sub Resources

We can also define resources that belong to other resources like so:

```
resources :pages do
  resources :comments, :trackbacks
  resource :author
end
```

This will create a full set of resource urls for the pages, comments, trackbacks and author controllers, which will pass the controller the necessary params to pull those models.

Namespacing

Lets say you have an admin section of your site. You might want to namespace some routes, like so:

```
namespace :admin do
  resources :pages
end
```

This will create the following routes.

```
    admin_pages GET    /admin/pages(.:format)           {:action=>"index",
:controller=>"admin/pages"}
                POST   /admin/pages(.:format)           {:action=>"create",
:controller=>"admin/pages"}
 new_admin_page GET    /admin/pages/new(.:format)       {:action=>"new",
:controller=>"admin/pages"}
edit_admin_page GET    /admin/pages/:id/edit(.:format) {:action=>"edit",
:controller=>"admin/pages"}
     admin_page GET    /admin/pages/:id(.:format)       {:action=>"show",
:controller=>"admin/pages"}
                PUT    /admin/pages/:id(.:format)       {:action=>"update",
:controller=>"admin/pages"}
                DELETE /admin/pages/:id(.:format)       {:action=>"destroy",
:controller=>"admin/pages"}
```

Note that our pages controller must be placed in the app/controllers/admin directory for it to work. We must also namespace our controller like so:

```
class Admin::PagesController < ApplicationController
```

Routing Exercise

In this exercise we will use rake to create routes and test them. Create a new Rails instance (or use hello world), and edit config/routes.rb

Named Routes

Create a log in route on one of your rails instances.

name: login

path: /login

action: new

controller: session

Create the corresponding log out route

name: logout

path: /logout

action: destroy

controller: session

Named routes with parameters

Create the following route:

name: search

path: /search/:search_term/category/:category_id

action: index

controller: search

Now create a search controller with an index method. Put a breakpoint in it, and verify that if you hit the url above, you do in fact hit that controller.

At the breakpoint, check the params hash and verify the values in it.

At the breakpoint, verify that the search_path method exists. You'll need to pass it 2 values.

Finally, also at the breakpoint, verify that the methods login_path, login_url, logout_path and logout_url have indeed been generated.

Answers

Named Routes

```
match "/login" => "session#new", :as => "login"
match "/logout" => "session#destroy", :as => "logout"
```

Named Routes with Parameters

```
match "/search/:search_term/category/:category_id" => "search#index", :as =>
"search"
```

Session, Request and Flash

Important hashes

Rails maintains several important hashes for you.

The Params Hash

Any values that are passed to a controller from the user are available in the params hash. This includes:

- data from the URL, eg www.myapp.com/page/1
- Get data appended to the URL after a question mark
- Post data submitted using a form or AJAX request.

The Request Hash

Information about the current request is held in the request hash. This includes the current URL, port, params, and a whole host of other useful bits and bobs.

The Session Hash

The Session Hash contains information about the current user session. By default, the entire session is held in an encrypted cookie.

If you prefer, you can change this so the cookie only holds a key, and the session itself is held in the database. This is useful if you need to store lots of data in the session.

The Flash Hash

The Flash Hash is where you put information which will be available to the next controller. For example, if you want to redirect, and show a success message, put the message in flash and it will be available to the next controller, and will then be automatically deleted.

Important Hashes Exercise

Lets take a look inside these hashes.

Start a server using:

```
rails s
```

Now add a breakpoint to one of your controllers. Trigger the breakpoint and inspect the hashes.

You can also view the contents of a hash in your view. Add a line like this:

```
<%= params.inspect %>
```

to a view, the whole params hash will be output.

Active Record

Active Record Relations

AR Relations are a relatively new Rails feature. Instead of pulling an array from the database and manipulating it within Ruby. AR Relations construct an SQL statement.

The SQL statement is executed only when Ruby objects are actually needed, at which point an array is generated and cached.

Where

Assume we have a Kitten model which extends ActiveRecord::Base

We can create an SQL where clause like so:

```
Kitten.where :id => 12
```

If we want a number of kittens we can pass an array like so:

```
Kitten.where :id => [1,2,6,12]
```

If we want something more complex we can write:

```
Kitten.where("created_at > ? and created_at < ?", Time.now - 2.weeks, Time.now -
1.week)
```

This will get all the kittens between 1 and 2 weeks old. Awww.

Method Chaining

Methods called on an AR relation object both affect, and return the AR relation object itself, so you can chain them together, constructing complex SQL queries using a friendly Ruby interface.

For example:

```
Kitten.where(:id => [1,2,6,12])
      .where("created_at > ?", Time.now - 2.weeks)
```

This will get any kittens with the specified ids and which are less than two weeks old.

Order

We can specify an order like so:

```
Kitten.order(:name)
```

Or if we need to execute some SQL:

```
Kitten.order("lower(name) DESC")
```

This will sort all the kittens in reverse name order ignoring case.

Of course you can chain order commands together as well:

```
Kitten.where(:id => [1,2,6,12]).order("lower(name) DESC")
```

Getting SQL

You can return SQL using the to_sql command. This is useful for debugging.

```
Kitten.where(:id => [1,2,6,12]).order("lower(name) DESC").to_sql
```

Returns:

```
"SELECT `people`.* FROM `people` WHERE (`people`.`id` IN (1, 2, 6, 12)) ORDER BY
lower(name) DESC"
```

Notice how the AR Relation is noticibly shorter, and just, well, nicer.

Getting an Array

Your AR relation object will be implicitly converted to an array which will be held by the ActiveRecord object as soon as you start to access it using the each method.

If you need to explicitly generate an array though, for example if you are doing some complex pagination, you can do so using the

```
to_a
```

method.

Getting a Blank Scope

To get a blank relation (which will return everything in the table) use a blank scope:

```
Kittens.scoped
```

will return all of the kittens.

Active Record Relations Exercise

AR Relations are a cool new Rails feature that lets you build up a query which will be executed only when you try to access Ruby objects.

Let’s have a go now.

Extend one of your classes

Extend your superhero class, or one of your other classes with a number of attributes, including for example:

- A title (string)
- A birthday (datetime)
- A friend count (integer)

The correct way to do this is by using a migration.

You can generate a migration using the rails generator command:

ruby script/rails generate migration migration_name

Now create a few objects, either using your web application, or directly in the database.

Where

Find every superhero that was created in the last 5 minutes.

Find every superhero, given a range of ids that was created in the last 5 minutes.

Sort

Sort your superheroes by birthday.

Sort your superheroes by in reverse birthday order.

Sort your superheroes by birthday, and then by name. (Note, use more than one sort command)

Search

Search for an object by passing an SQL like statement to the where method as a string. Find all the superheroes who have the letter a in their name.

Implement a parameter based sort and search

Modify one of your index methods in a controller such that it can accept a :sort parameter and a :search parameter.

Create an AR relation object which searches for the value passed in by the search parameter.

Use a case statement to modify your AR relation with a sort statement, if params:sort is present.

Return the AR Relation to your view and output the result.

Finally create a form that submits to your index method on the controller and passes search and sort parameters.

Very nice.

Scopes

If you have a query you need to run often, you can declare default AR relations on a model. These are called scopes.

Declaring a Scope

Say you have a bunch of users, and some of them new to the site.

You might find all the new joiners using something like this:

```
User.where("created_at > ?", 1.week.ago).order("created_at DESC")
```

All fine and good, but a lot of typing if you use it often. We can create a scope on the model like this:

```
class Article  < ActiveRecord::Base
```

```
  scope :published, where(:published => true)
end
```

We can now call:

```
User. published
```

…to return the AR relation.

Limitations

Scopes have one important limitation, they cannot accept parameters, so you might set up a scope to return all the published articles for example, but you couldn't create one to return all the articles with a particular author which you would pass in.

If you want to pass a parameter, create a static method instead which returns an AR relation.

Chaining Scopes

Because a scope returns an AR relation, you can chain them together, for example:

```
class Article  < ActiveRecord::Base
  scope :not_published, where(:published => false)
  scope :not_spam, where(:spam => false)
end
```

We could now do:

```
Article.not_published.not_spam.sort("created_at DESC")
```

This handy shorthand is much more readable and maintainable than sprinkling where clauses throughout your code.

Testing

Testing is a big deal in Rails. Rails encourages you to test your own code, to write acceptance tests as you go, and to code against the tests.

There are several methodologies. In this section we look at RSpec, Cucumber and Capybara.

Older Syntax

The RSpec syntax had a major upgrade in 2013. The older syntax had non-descriptive method names, for example “it” and “should”. This section covers the old style syntax.

Why Test?

There are three main reasons.

One, it reduces bugs. Test your code and you know it works. You know that feeling when you go home at night and you worry you might get a call that the server’s down? You won’t get that anymore.

Two, it makes refactoring a doddle. Need to upgrade a library? Run the tests and it’ll tell you exactly what’s broken.

Three, it makes you a better coder. Testable code is modular and clean. It has a sensible API. Methods are single purpose and parameterised. If you’re looking at your code and you can’t work out how to test it, chances are it’s probably not very good code. When we test code we are forced to use it. It makes us think.

TDD

TDD is Test Driven Development. The idea is that we write our tests first, then write the code to make them pass. It can be a tricky discipline to master, but once you have, it’s actually incredibly helpful.

On the train home sometimes I go to look at what I’ve done that day, and I don;t have a running server. I’ve been working against tests all day.

It feels good to build something that definitely works.

BDD

Behaviour Driven Development is a methodology that encompasses TDD, and also Acceptance Test Driven Planning, in which tests are used as a specification.

The Failing Test

Because we write our tests first, our tests generally start out by failing. We then write the code to make them work. When we come up with a new feature, we can write tests for that new feature first, then fill in the minimal code to make the tests pass.

Generally speaking, we tend to write a few tests, then a bit of code, then a few more tests, and another bit of code.

Installing RSpec

RSpec is a gem. add it to your Gemfile

```
gem 'rspec'
```

Test your installation by typing

```
rspec -v
```

Rspec should respond

Writing a spec

Lets write tests for our kitten class. I'll assume we're starting a new class here, but if you already have a kitten, don't delete it.

```
kitten.rb

class Kitten
end
```

Now lets create a test file for it

```
kitten_spec.rb

require "user.rb"
describe Kitten do
end
```

Running the spec is simple, just type:

```
rspec kitten_spec.rb
```

If all has gone well, we should get:

```
rspec kitten_spec.rb
No examples found.

Finished in 0.00011 seconds
0 examples, 0 failures
```

Describe and It

"describe" and "it" are the first two methods we are interested in.

The "describe" method creates an instance of Behaviour, which knows about our class and accepts a block of tests.

The it method returns an instance of class Example. It creates a test.

Creating an example

Kittens should be happy. Lets check to see if our kitten is.

```
require "kitten.rb"

describe Kitten do
  it "should be happy" do
  end
end
```

“It” receives a string, the name of the test, and also a block which contains the test.

Run it now, you’ll see it works. If you want nicer output, you can use a formatter, like so:

```
rspec kitten_spec.rb --format documentation

Kitten
  should be happy

Finished in 0.00032 seconds
1 example, 0 failures
```

Writing the test

This test isn’t terribly good, it isn’t testing anything. Lets add a test. A test is simple, it just sets up the environment, does something to the object, then tests to see if the result is as expected. It’s simple input, process and output.

```
require "kitten.rb"

describe Kitten do
  it "should be happy" do
    kitten = Kitten.new
    kitten.should be_happy
  end
end
```

be_happy is a magic method. It’s a Matcher that RSpec has created for us. It calls the happy? method on kitten, and passes the test if it returns true. As well as be_, we can also use “be_a*“ and ”be_an*”, if they read better. They’re all equivalent.

Try it now.

```
Kitten
  should be happy (FAILED - 1)

Failures:

  1) Kitten should be happy
     Failure/Error: kitten.should be_happy
     NoMethodError:
       undefined method `happy?' for #<Kitten:0x1041bf760>
     # ./kitten_spec.rb:6

Finished in 0.0004 seconds
1 example, 1 failure
```

You see, RSpec is telling us about a problem. The happy? method doesn’t exist. This is called a failing test.

Satisfying the test

To pass this test we write the minimal possible code to make it pass. Let’s return to our kitten object:

```
class Kitten
  def happy?
    true
  end
end
```

Now our kittens are happy. If we run the test, all is well.

Challenge

Here’s a challenge for you. Our kittens are only happy if they have mittens, where mittens is an instance variable.

Write an example that tests that our kittens are not happy without their mittens.

Write another example that tests our kittens are happy once they have their mittens (kitten.mittens = true)

Hint, use kitten.should_not be_happy to test for a negative.

Matchers

Matchers are methods that match conditions in a test. We can make these up on the fly, RSpec will create them.

```
be_happy
```

is an example of a matcher. We can use matchers to test for the presence of a key in a hash, we can test to see if our result is an instance of another class, we can even write custom matchers.

Check the documentation here for some more examples of matchers:

```
http://rubydoc.info/gems/rspec-expectations/2.4.0/RSpec/Matchers
```

Equality

We can test for equality. Let’s see if our kitten is numerate:

```
require "kitten.rb"

describe Kitten do
  it "should be numerate" do
    1.should eql(1)
    (1+1).should eql(2)
  end
end
```

Here we return true if the result of the first expression is equal to the second.

Of course we’re not actually testing our kitten here.

Challenge

Write a “should be able to multiply” expectation for your kitten. Write a do_multiplication_sum method that satisfies the tests.

Further Work

Write specs for your fluffy animal or lethal warship class. Try and achieve 100% test coverage.

Mailers

Most Rails applications at some point will want to send email. Even if you have a separate mail spooler, you are still likely to want to send admin notifications, password recovery links, etc.

Sending emails from Rails is simple.

Generating a mailer

Create a mailer using the Rails generator command.

```
rails g mailer my_mailer
```

You will get a new file in the mailers directory, and a new sub-directory in the views folder with the same name.

Mailer code

A mailer is just like a controller, only instead of calling the render method, it calls the mail method.

Let's create a welcome message. We'll add a default :from line which sets the default sender. You can set other defaults as well, such as a default reply_to.

Then create a method called welcome_message. This is the method we will call to send our email.

```
class PostmarkMailer < ActionMailer::Base

  default :from => "administrator@example.com"

  def welcome_message(email)
    @email = email
    mail(
      :subject => 'Welcome to Streetbank',
      :to      => email
    )
  end

end
```

As you can see, it's almost trivially simple to do.

View

To send your email you must create at least one view. In the views/my_mailer directory create a file called:

```
welcome_message.html.erb
```

In it put any erb you like. You have access to the @email variable because we declared it in the MyMailer#welcome_message method.

```
<p>
```

```
  Hello <%= @email %>! Welcome to the site.
</p>
```

Creating a text only view

Some people prefer a plaintext email. You can provide this easily by creating a second view, like so:

Create a file called:

```
welcome_message.text.erb
```

Now add code, somthing like this:

```
Hello <%= @email %>! Welcome to the site.
```

Your mail will now be sent as a multipart email with a plain text component and an html component. Your user will be able to choose which one they see from within their email client.

Sending the mail

You send the email by creating an instance of the mail object, and calling deliver on it, like so:

MyMailer.welcome_message(“email@example.com”).deliver

Simple!

letter_opener gem

In your development environment you’re unlikely to want to send real emails, but you do want to see what your mailer is generating.

The letter_opener gem is useful here.

In your Gemfile:

```
group :development do
  gem "letter_opener"
end
```

This includes this gem in the development environment only.

Useful Gems

Some gems are used so frequently they almost become part of the core. Here are a few:

Foreman Gem

Say you have an app that requires multiple processes to be running. Starting all those processes separately is a pain.

The Foreman gem is used to start and manage multiple processes with a single command. You would typically use it on your development server to quickly get set up for development.

For example, you might need a Delayed Job worker running in addition to your Rails instance. Foreman can start these processes at the same time.

Define all your processes in a Procfile in the root of your project. The Procfile contains the name of the task, followed by a colon, and the command to start the process, like so:

```
web: bundle exec unicorn -p$PORT -c ./config/unicorn.rb
worker:  bundle exec rake jobs:work
```

(note, Foreman uses port 5000 by default)

To start all these processes simply type:

```
foreman start
```

Heroku will also use the Procfile for managing default processes. For example:

```
heroku ps:scale worker=2
```

actually inspects the Procfile for an entry called worker, and runs the associated command.

Active Admin

Active Admin is a gem that allows us to create a beautiful, fully featured admin system quickly and easily. You can make something pretty amazing with almost no code at all.

It’s still in an early version and is not 100% stable between releases. Be prepared to submit the odd bug fix, to read commit logs, and to monkeypatch any errors.

You can find out more at: activeadmin.info

Pagination

Pagination links are those tiny little links at the bottom of a page which get you to the next page or results. Pagination is a solved problem in Rails.

Choose between the following.

Will Paginate

Super simple drop in gem. Pass a page parameter to your finder, and it’ll modify the AR relation for you.

The documentation is here:

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

Kaminari

A newer pagination gem which runs as an engine. It does pretty much the same thing as Will Paginate.

Here are the docs:

https://github.com/amatsuda/kaminari

Paperclip

Paperclip is the gem that's commonly used for file uploads. Attachments are treated just like any other attribute, and are not saved till validation has been passed.

The documentation is here:

https://github.com/thoughtbot/paperclip

Delayed Job

The Delayed Job gem allows you to shift slow or blocking processes off the main Rails stack, and onto a worker process. The most common use for this is email. Say someone has added a site, and you need to email a notification message. It's not that important, and you don't want to block the Rails process while you send it.

You would normally send the message like this:

```
MyMailer.notification_message( person ).deliver
```

Instead you could do this:

```
MyMailer.delay.notification_message( person )
```

The message will be sent from a separate process.

Documentation is here:

https://github.com/collectiveidea/delayed_job

Faye

Long polling is a technique in which a network connection is opened by a client, and kept open until the server is ready to push an update.

Say you have a web application in which

RAils is really

Asset Pipeline

Your asset pipeline is the conduit through which you import stylesheets, JavaScript files, images, etc.

Namespacing CSS

Because all your stylesheets are imported with each request, you have to be clever not to apply page specific rules everywhere. I typically add the controller and model as a class to the body element. This allows me to namespace my SCSS.

app/helpers/application_helper.rb is a module that's imported by all your views. You can define helpful methods in it. The following method will create a string based on your controller, action, and optionally a version parameter. Add it to the application helper, or modify it your purposes.

```
def body_class
  bc = []
  bc << params[:controller].gsub('/', ' ')
  bc << params[:action]
  if @version
    bc << "version_#{@version}"
  end
  bc.join(' ')
end
```

Now in your app/views/layouts/application.html.erb file, add the new class:

```
<body class="<%= body_class %>" >
```

In your SCSS files you can now write things like:

```
.home.index {
  h1 {
    font-size:5em;
  }
}
```

Asset Digest

A Digest is a strings which are added to the end of your asset filename. It's created by hashing the file. If any of your files change, the digest is updated, and so the filename is updated. This gets round caching issues.

Turn on digests like this:

```
config.assets.digest = true
```

Now view source. You'll see all your assets are compressed.

Asset Compilation

Let's have a look at asset compilation:

By default your development server will serve all your assets as separate files. Turn off asset debugging in config/environments/development.rb by modifying the line:

```
config.assets.debug = true
```

to:

```
config.assets.debug = false
```

restart your server to see the results.

Advanced Rails Topics

Initialisers

Initialisers are ruby files that are run once before a Rails server starts up. They live in the config/initialisers folder.

Use them to:

- Monkey patch anything you need to.
- Configure gems that require setup.
- Initialise any global variables you may require.

Because they are run once before the Rails instance starts, if you make a change, you need to stop and start your server or console.

Railties

http://blog.engineyard.com/2010/extending-rails–3-with-railties

Security

In this section we'll look at the 3 most common attack vectors and what you can do about them. We'll also look at:

- SQL injection attacks
- Cross site scripting attacks (XSS)
- Mass assignment

We'll also look at patching your Rails version, and how to get alerted about any issues.

SQL Injection Attacks

An SQL injection attack may occur when you take a user submitted string and use it to form a SQL query.

Rails features pretty good protection from this type of attack, but you have to know how to use it.

The Attack

Let's say you do this:

```
Person.where( "id = #{params[:id]}" )
```

And someone hits a url something like:

```
www.myapp.com/person/1;DROP TABLE users
```

The value of params[:id] is now "1;DROP TABLE people"

This will allow a hacker to drop your users table.

Defending against it

This is easy to counter. Change your code like this:

```
Person.where( "id = ?", params[:id] )
```

The ? will be substituted for an escaped version of params[:id]

You could also do this:

```
Person.where( :id => params[:id] );
```

Never use string manipulation to compose a query from user submitted data. Mistrust everything the user submits.

Cross Site Scripting (XSS)

A cross site scripting attack occurs when a user submits some JavaScript, perhaps in a comment. The app then inserts that JavaScript onto the page. Other users will view the page and execute the malicious JavaScript.

The Attack

A user goes to your page and, in a comment box, types a string containing JavaScript.

Your app shows the comment to other users who will now execute that JavaScript.

The script might steal their password or session cookie, it might redirect them to another site, it might replace your site with a game of Asteroids depending on the hacker's motivations.

Bear in mind that there are many ways to execute a script other than using script tags. You might add an onload attribute to an image for example. It's not enough to simply filter out script tags.

Defending against it

All strings that you compose are escaped by default when output by erb. If you want to output an unescaped string, call html_safe on it, eg:

```
"<p>Hey there</p>".html_safe
```

Be sure to escape all user submitted content.

Mass Assignment

Mass assignment is a technique that was used against early Rails apps.

It would take advantage of the update_attributes method, using it to update attributes on a model.

The Attack

The attacker would compose a URL like this and submit via a post request to call the update method.

```
www.myapp.com/person/1?password=12345&confirm_password=12345
```

Password and confirm password would then be saved in the params hash, and the person's password would be changed.

Defending against it

We use attr_accessible to define a whitelist of attributes that can be accessed by mass assignment.

```
attr_accessible :email, :name
```

Only attributes on the list can be set via update_attributes.

If we want to set the password we must do it explicitly like this:

```
@person.password = params[:password]
@person.save
```

You would want to make sure you had verified the person before updating their password.

Patching Rails

Rails is fast moving, and you’ll want to keep up to date. Because Rails is packaged as a gem, you can update it simply by updating your Gemfile. A good test suite helps here.

Hearing about exploits

Occasionally exploits are discovered. When they are, you’ll want to hear about them fast.

I would suggest joining the Ruby Weekly mailing list:

Ruby Weekly: http://rubyweekly.com/

If you’re interested, you may also wish to join the Ruby Lang mailing lists:

Ruby Lang: http://www.ruby-lang.org/en/community/mailing-lists/

Scaling Rails

“Rails doesn’t scale” is a common accusation. After all, Twitter was originally written in Rails, and the Fail Whale was a common sight.

Nowadays Rails can actually scale quite well.

Scalable cloud hosting

The simplest way to scale Rails it to provide multiple servers, and the easiest way to do this is with a cloud host.

Heroku is a very good option. They provide scalable cloud hosting solutions. You can provision extra Rails instances within a few minutes by dragging sliders.

Using an asset host

You’ll want to shift as much of the load as possible off your server. Using an asset host to host your static files is a very good solution.

Amazon S3 storage integrates nicely with Rails. You can configure Paperclip to use it by default.

Cloudfront is another good option. Tell Rails to use Cloudfront as an asset host and it will map the URLs of all your static assets to the Cloudfront domain. Cloudfront will then download your assets and serve them for you. It’s an extremely simple one shot solution to scaling.

Caching with MemCache

When you render a page, it;s likely that much of the page will remain the same for some period of time. Let’s say you have a list of users, that list page will remain the same until a new user signs up.

Memcache is a key value store. It can hold page fragments in memory so they don’t need to be re-rendered. Because everything is in memory it’s blazingly fast. Fragments can be invalidated if something changes, and memcache will automatically update the cache to take account of it.

Publishing an API

Because Rails separates controllers and views, Rails allows you to easily render your pages in another format. JSON for example.

Say you have an app which responds to a url like:

```
localhost:3000/page/1
```

By default the template which would be rendered would be:

```
app/views/page/show.html.erb
```

Now hit the URL again, like this

```
localhost:3000/page/1.json
```

Now the template rendered would be:

```
app/views/page/show.json.erb
```

We can provide a different template, a different view on the same data.

Serialising to JSON

Rails allows you to easily serialise any Active Record object, or AR relation to JSON.

If you're using 1.9.2 or above, you can convert hashes and arrays to nested JSON objects just using to_json.

```
{a: [1,2,3], b: 4}.to_json
```

In Rails, you can call to_json on Active Record objects. You can pass :include and :only parameters to control the output:

```
@user.to_json only: [:name, :email]
```

You can also call to_json on AR relations, like so:

```
User.order("id DESC").limit(10).to_json
```

You don't need to import anything and it all works exactly as you'd hope.

Serialising to XML

You can also serialise to XML using the to_xml method. This works nicely and will output a nicely formatted

Accessing an API

In this exercise we're going to look at querying an API. We're going to use the Ruby Standard Library to do this. Find the documentation here:

```
http://www.ruby-doc.org/stdlib-1.9.2/
```

Her Gem

The Her gem is a seamless object relational mapper for use with a REST API. Active record like method calls are converted into http requests to the API.

JSON

Net

We're going to use two libraries, Net and Json. First lets look at Net. As you might expect, this library allows you to pull files from the internet. Click on Net, and then Net:HTTP. This class allows you to connect to a URL via HTTP and pull the contents.

```
require 'net/http'
require 'uri'
def get_data
  result = Net::HTTP.get URI.parse('http://www.google.com')
  puts result
end
```

As you can see, here we are pulling the contents of Google.com.

Try this now.

The Graph API

Facebook exposes a particularly lovely restful API, called the Graph API. Check it out here:

```
http://developers.facebook.com/docs/reference/api/
```

From the graph you can pull any bit of data about anyone, with some restrictions, and subject to permission.

Here's my public information:

```
http://graph.facebook.com/nicholas.a.h.johnson
```

Exercise

Pull this data now using Ruby and have a look…

Json

Next we're going to parse our data. We'll need to use the JSON Library, so include it now.

```
require 'net/http'
require 'uri'
require 'json'
def get_data
  result = Net::HTTP.get
URI.parse('http://graph.facebook.com/nicholas.a.h.johnson')
  JSON.parse(result)
end
```

the data is parsed into a hash, which we can use as we like.

Exercise

Create a Rails instance that receives a Facebook ID or username and outputs all the data about that person on the screen.

Extension

The Twitter API is available here:

```
https://dev.twitter.com/docs/api
```

Extend your app so it can also pull a live Tweet stream for a user.

Further Extension

You can get a single user Twitter access token here:

```
https://dev.twitter.com/docs/auth/tokens-devtwittercom
```

Authorise yourself, and use your access token to post to your own timeline.

XML APIs with Nokogiri

XML is like violence - if it doesn’t solve your problems, you are not using enough of it. - nokogiri.org

Nokogiri is an XML parser with support for XPath and CSS3. It’s fast and super easy.

Installation

Install nokogiri using RubyRems

```
gem install nokogiri
```

Usage

```
require 'nokogiri'
require 'open-uri'

url = "http://www.myurl.com/feed.xml"
doc = Nokogiri::HTML(open(url))
```

We can then search for nodes by CSS:

```
doc.css('h1').each do |heading|
  puts heading.content
end
```

Or XPath

```
doc.xpath('//h3').each do |link|
  puts link.content
end
```

Pulling Data from the BBC

Let’s Use Nokogiri to pull a feed from the BBC. The BBC makes it’s program listings available in a variety of formats including XML. See here:

```
http://www.bbc.co.uk/programmes/developers
```

Pull an XML feed of upcoming programs.

Now write a rails application that displays a list of everything that is on today.

Gems and Version Management

Package management in Ruby is achieved using Gems. Gems are little bits of code. Typically they are written in Ruby, but may be written in any other language which the OS can compile. C++ is a popular choice.

A gem can make a module available, can monkeypatch an existing class, add a new class, or can do anything else Ruby can do.

RubyGems

RubyGems is a Gem manager for Ruby which pulls gems from a gem source, usually rubygems.org.

It is built into Ruby as of version 1.9.

A Ruby gem is a directory containing Ruby, YML and sometimes another language such as C++ which can be compiled. Gems are automatically downloaded from gem sources.

The various components of Rails are defined as gems, and Rails installation is accomplished using RubyGems.

Installing a gem

```
gem install mygem
```

Uninstalling a gem

```
gem uninstall mygem
```

Listing installed gems

```
gem list
```

Listing available gems

```
gem list --remote
```

Listing your sources

```
gem sources
```

by default this will be rubygems.org

Updating a gem

```
gem update mygem
```

Updating RubyGems

```
gem update --system
```

Bundler

Bundler is your application’s gem manager. Bundler maintains a list of gems which your application needs to run, plus their dependencies.

Bundler has two files:

- Gemfile - a user editable list of Gems
- Gemfile.lock - a generated list of specific instructions to the Bundler.

Gemfile

Configure Bundler using the Gemfile file. This file contains a list of all the gems your application needs to run, and optionally their versions.

Gemfile.lock

Gemfile.lock is automatically created for you based on your Gemfile when you run bundle install or bundle update.

It contains a list of all the gems currently in use, their specific versions, and all dependencies. You can use it to make sure that the same gems are used on all your instances, including your live server.

Check it into you version control, to synchronise gem versions.

Install

Install the gems in Gemfile.lock on your system or another system using:

```
bundle install
```

If there is no Gemfile.lock, it will be created for you.

If a file is listed in Gemfile, but not Gemfile.lock, Gemfile.lock will be updated.

Update

Update all Gems to the latest version based on your Gemfile using.

```
bundle update
```

Update a specific gem (for example the activeadmin gem) using

```
bundle update activeadmin
```

Gemfile.lock will be updated. Be sure to check it back into version control. No one else will be able to start a server until they run:

```
bundle install
```

RVM

If you have more than one project on the go, you may find that you need

Ruby Version Manager is a small piece of software that allows you to easily switch between versions of Ruby, and different gemsets for different projects. This is useful if you’re working on more than one project at once. Rubies are installed in your home directory.

Installing Rubies

Installing Rubies

```
rvm install 1.9.2
```

You can install multiple versions of Ruby including IronRuby, JRuby and MacRuby.

Picking a Ruby

Then you choose which one you want to use:

```
rvm use 1.9.2
```

Verify the new ruby is being used:

```
which ruby
/Users/nicholas/.rvm/rubies/ruby-1.9.2-p290/bin/ruby
```

Gemsets

Gemsets are sets of installed gems that you can switch between. Like Rubies, they are installed in separate directories in your home By default, installing a new version of Ruby also installs a new gemset. When you switch to a version of Ruby, you also switch to it’s gemset.

You can create additional gemsets using

```
rvm gemset create my_gemset
```

You can then use that gemset using

```
rvm use 1.9.2@my_gemset
```

Note

Note that although you get complete separation of Rubies and gemsets, RVM is not magical. You don’t get total separation between environments. Rake is global, if you upgrade it, it’s upgraded everywhere. Likewise your database.

RbEnv

Rbenv is a relatively new Gemset manager. You get similar functionality but it’s lighter weight and more Unixy. Rbenv works by defining shivs with the same name as the ruby commands. These shivs are declared at the start of the path, so when you type ruby into a terminal, the shiv is called instead.

The shiv looks for an RBENV_VERSION environment variable and uses it to choose which version of Ruby to use. If none is found it looks for an .rbenv-version file, first in the current directory, then recursively up until it reaches the root. Nice and simple.

Installing Rubies

Installing Rubies is a little more involved as you must download, unpack and build them in the correct directory.

You can find instructions for doing this on the rbenv github page:

```
https://github.com/sstephenson/rbenv
```

There’s another related ruby-build project that simplifies loading rubies

```
https://github.com/sstephenson/ruby-build
```

Gemsets

RBenv doesn’t have gemsets by default, but there are plugins to enable this feature.

rbenv-gemset is a good example:

```
https://github.com/jamis/rbenv-gemset
```

Difference between RVM and RBEnv

RVM works by overriding several built in command line scripts, including cd. Rubyists tend to be comfortable with overriding and augmenting basic types, because Ruby makes it so easy and the community encourages it. However Unix people are less than comfortable applying the same principles to shell scripting.

RBenv on the other hand is less intrusive. It stands between you and Ruby, but doesn’t override any unix. For this reason many people prefer it, though it is a newer project, and not quite so easy to use.

Creating a Gem

A gem is a directory of code, typically, though not always Ruby code, plus a little metadata to help RubyGems and Bundler to manage it nicely.

In this section we’re going to create a simple gem using Bundler.

Creating a gem using Bundler

We are going to create a summarise gem which will extend the string class with methods to create a summary, and to check whether the string can be summarised, like so: class String def summarise(l=200) i = 0 self.split.map{ |word| word if (i += word.length) < l}.compact.join(’ ’) end

```
  def summarisable?(length=200)
    return self.summarise(length) != self
  end
end
```

first of all create the gem:

```
bundle gem summarise
```

We now have a new summarize directory containing a lib directory for code, a gemspec file for metadata, and a few other files.

We also have an empty git repository initialised for us.

Gemspec

The gemspec file is the heart of your gem, it contains all the metadata about your gem. It should look something like this:

```
# coding: utf-8
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'summarise/version'

Gem::Specification.new do |spec|
  spec.name          = "summarise"
  spec.version       = Summarise::VERSION
  spec.authors       = ["Nicholas Johnson"]
  spec.email         = ["email@domain.com"]
  spec.description   = %q{TODO: Write a gem description}
  spec.summary       = %q{TODO: Write a gem summary}
  spec.homepage      = ""
  spec.license       = "MIT"

  spec.files         = `git ls-files`.split($/)
  spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
  spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
  spec.require_paths = ["lib"]

  spec.add_development_dependency "bundler", "~> 1.3"
  spec.add_development_dependency "rake"
end
```

Notice how spec.files is set by getting the files currently included in git.

Also notice how the version is taken from the Summarize::VERSION constant. This is defined in lib/summarise/version.rb. The version can be updated by editing this file.

Gem Contents

We now define the gem in the lib directory.

lib/sumarise.rb

This file simply imports the rest of the code.

```
require "summarise/version"
require "summarise/string_extensions"
require "summarise/string"

module Summarise
end
```

lib/summarise/string_extensions.rb

This declares the methods we want to add.

```
module Summarise
  module StringExtensions
    def summarise(l=200)
      i = 0
      self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
    end

    def summarisable?(length=200)
```

```
      return self.summarise(length) != self
    end
  end
end
```

lib/summarise/string.rb

This extends the String class.

```
class String
  include Summarise::StringExtensions
end
```

Building the gem

Because the gemspec uses Git to discover which files to include, we must first commit your updated files.

```
git add .
git commit -a -m "first commit"
```

Build the gem using the gem build command:

```
gem build summarise.gemspec
```

You will create a file called something like: summarise–0.0.1.gem

Local Private Gems

If you want to keep your gem private, you can deploy it directly into your vendor/gems directory, like so:

First unpack it into vendor/gems:

```
gem unpack summarise-0.0.1.gem --target /path_to_rails_app/vendor/gems/.
```

Now declare it in your Gemfile:

```
gem 'summarise', :path =>
"#{File.expand_path(__FILE__)}/../vendor/gems/summarise-0.0.1"
```

Finally install it into Gemfile.lock

```
bundle install
```

This is a good way to develop a gem, as you can deploy it locally and work on it in situ.

Uploading your gem to RubyGems.org

If you'd like to share your gem with the community, you can also push it to RubyGems.org

You'll need a RubyGems account:

```
https://rubygems.org/users/new
```

Now simply push the packaged gem:

```
gem push summarise-0.0.1.gem
```

Plugins and Engines

Rails also allows you to define and install plugins. Plugins sit in the vendor directory and perform some useful function. They are automatically included when the rails instance starts up.

As of Rails 4 though, plugins are no longer supported. Gems or engines are used instead.

Engines are new and allow a full MVC stack in a plugin. A deep discussion of plugins and engines is beyond the scope of this course, but do read up on them if you’re interested.

Additional Exercises

New Programmer Exercises

Don’t walk before you can swim. These Ruby Newbie exercises are designed to fill in any gaps in your knowledge. If you’re struggling further on, try these first. If you’re happy, skip to the next section.

Exercise 1 - If/Else

Write a program that accepts a two numbers and tells the user which one was largest.

Use the gets method to accept keyboard input:

```
gets input_number_one
```

and the to_i method to convert keyboard input into an integer like this:

```
input_number_one.to_i
```

Your program should accept two inputs, then use if, elsif and else to tell the user which was largest, or if they were equal.

Put your program logic into a function that you can call.

Exercise 2 - The Case Statement

Write a food suggestion engine that listens to the user’s food desires and suggests a food to match. You may wish to use a case statement.

Hint: When using gets, you may need to use the chomp! string method to remove trailing newline characters. Make this as clever as you want.

Exercise 3 - Functions

1. Write a function that accepts an integer and returns the value of the integer * 2.
2. Write a function that accepts two integers and returns the value of the two integers added together.

Exercise 4 - Looping Exercise

1. Use a for loop to print the numbers 1 to 10 backwards
2. Use a for loop to print the number one once, the number two twice, etc.

Bonus Credit

1. Implement the above programs using an until loop.

A lightning quick CMS with login

Rails is really great at content management. In this section we’ll create a simple content management system.

Creating the Application

From a terminal type

```
rails new my_cms
```

and move into the my_cms directory.

The Model

The first part of the MVC triangle is the model. Let's create a model now, and hook it up to the database.

Creating the Page Model

Models hold all the information about assets in our application. The Page class will define he actions we can take on a page in our CMS. We can then create page objects based on this class.

In the app/models directory create a file called Page.rb containing the following code:

```
class Page < ActiveRecord::Base

end
```

We don't need anything in it yet. By extending ActiveRecord::Base we have gained all the functionality we need to get page information from the database.

That was easy wasn't it.

Configuring the Database

In this part of the course tutorial we'll be using MySQL because it's fast, free and universally available. Rails will happily talk to other databases too, as we have seen earlier.

open the file called config/database.yml (pronounced Yamel) and edit it to look something like the following:

```
development:   adapter: mysql
   database: my_cms_development
   username: root   password: password
   socket: /var/run/mysqld/mysqld.sock
test:   adapter: mysql
   database: my_cms_test
   username: root
   password: password
   socket: /var/run/mysqld/mysqld.sock
production:   adapter: mysql
   database: my_cms_production
   username: root
   password: password
   socket: /var/run/mysqld/mysqld.sock
```

This file defines access details for 3 databases which are used when our application is running in development, test or production mode. We'll only need one of them initially, the development database.

The settings are:

- adapter: the database type eg. sqllite or mysql
- database: the name of the database
- username: usually root
- password: the password for the account
- socket: the path to mysql on your local machine. You may not need this line on a mac.

Making changes to the configuration files is one of the few occasions when we need to restart the server so if your server is running stop it with:

```
ctrl-c
```

and start with:

```
ruby script/rails server
```

Creating the Database

Next we'll need to create our database.

To create the database this we'll use a tool like CocoaMySQL (Mac) or SQLYog Community Edition (Windows). If you're comfortable using the command line this is fine too. Some people like to use PHPMyAdmin when working on a server.

If you're using one of the GUIs creating the database is as simple as clicking the plus button or choosing an option from a menu. Create a database called my_cms_development.

Migrations

Migrations provide a clean shorthand way to set up tables in our database. Each migration has a timestamp and two methods: up and down. When we initialise a database Rails runs all the up methods on all the migrations in the db/migrations directory in turn. By using the down method we can rollback our database to an earlier state.

Rails stores the timestamp of all migrations which have been run. This is useful when working in team. If someone writes a new migration we can easily bring our database up to date. It's also useful when updating a live database. We can go from any state to any other state without loosing data.

Creating the Migration

At a command line type:

```
ruby script/generate migration create_pages
```

This generator automatically make a migration for us using the correct timestamp.

Edit the new file. It will be called something like: db/migrate/20081107103903_create_pages.rb

```
class CreatePages < ActiveRecord::Migration
  def self.up
    create_table :pages do |t|
      t.string :title
      t.text :introduction
      t.text :text
      t.timestamps
    end
  end
  def self.down
    drop_table :pages
  end
end
```

The short up method will generate all the SQL we need to create a pages table in our database. The down method generates the SQL to remove the table.

Running the migration

Rails comes with a handy utility called Rake. Rake is a Ruby version of a Linux program called Make. It's used to automate common tasks.

We'll use Rake to set up our database tables. From a terminal type:

```
rake db:migrate
```

This task executes the up method on all the files in the db/migration directory. If it runs smoothly check your database to see your new table. If there was a problem run the task again with the –trace option to get more detailed debugging information:

```
rake db:migrate --trace
```

Other useful rake tasks

```
rake db:migrate VERSION=<timestamp>
```

migrates up or down to a specific timestamp.

```
rake db:migrate:redo
```

redoes the last migration, calling down, then up on it.

```
rake db:migrate:reset
```

drops and reloads the entire database.

```
rake -T
```

lists all available rake tasks.

Check your Database

Using Sequel Pro (Mac), SQLYog (Windows), or whatever you have to hand, view your database. You should see something like the following:

The pages table has been created. Notice:

- An id field has been created. This is auto incremented and unique.
- title, introduction and text fields have been created. These are all of the correct type.
- created_at and updated_at timestamps have been added.

The Controller

The second thing we need is a controller. Our route will call methods on the controller.

Create the Pages Controller

Create a file called app/controllers/pages_controller.rb. Add an index action to it.

```
class PagesController < ApplicationController
  def index
  end
end
```

The View

Views are rendered by the controller. Views get access to all of the controller's instance variables, plus the params hash. The default place to put a view is in a file with the same name as the method, in a directory with the same name as the controller.

Create the Pages index view

Create a file called app/views/pages/index.html.erb. This will be our default page view. We'll extend this later but for now lets just put a title on it so that we know it's working.

```
<h1>Pages</h1>
```

The Route

We've played with routes already so this should be a doddle.

Create the route

Rails is optimised for CRUD, so we can create a whole set of CRUD routes at a stroke. Open up config/routes.rb and add the line:

```
resources :pages
```

We can view our routes from a terminal. Type:

```
rake routes
```

for a full listing. We now have routes for our CMS.

Test the server

Start the server with

```
ruby script/server
```

View the server at http://localhost:3000/pages. You should see something a nice web page.

Creating Pages

In order to create pages we're going to need a few things:

- A web form to collect information about the page
- A controller action to display the web form
- Another controller action to save the page

The New Action

To let us create pages we'll need to add a couple of actions: new - to render the new page form, and create - to actually do the saving of the page.

Edit the app/controllers/pages_controller.rb file as follows (new lines are bold)

```
class PagesController < ApplicationController

  def index
  end

  def new
    @page = Page.new
  end

end
```

The New View

We now need to create a form to allow us to Create our page. We'll use several helpers to do this. Create a new file called /app/views/new.html.erb:

```
<h1>New page</h1>

<% form_for @page, :url => {:action => 'create'} do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :introduction %><br />
    <%= f.text_field :introduction %>
  </p>

  <p>
    <%= f.label :text %><br />
    <%= f.text_area :text %>
  </p>

  <%= f.submit "Create" %>
</p>
<% end %>
```

Notice the form helper accepts a block and passes f to it, f being the form object.

Checking the Form

Visit localhost:3000/pages/new.

The form is not functional at the moment as we have not yet written an action to save the data

Saving the Data in the Create action

In order to save the data we need to add another action to our pages controller like so:

```
class PagesController < ApplicationController

  ...

  def create
    @page = Page.new(params[:page])
    if @page.save
      flash[:notice] = 'Page was successfully created.'
      redirect_to:controller => 'pages', :action => 'index'
    else
      render :action => "new"
    end
  end

end
```

This action attempts to save the page to the database. If the save is successful it redirects to the index action. If it isn’t, it re-renders the new form.

If the save is successful the action writes to a special variable called flash. This variable is available to the next action that gets called. It’s a lightweight way of communicating between actions. Here it just tells the next action that the page was created successfully. We’ll display the contents of the flash variable to the user later.

We should now be able to save our data. Click save and then check the database. You should see that your page has been saved and you should be redirected to the index view.

Extending the Index View

It would be great if the index view could render a list of pages. Let’s add this feature. First we’ll need to get a list of pages so extend the pages controller like so:

```
class PagesController < ApplicationController

  def index
    @pages = Page.all
  end

 ...

end
```

Because Page extends ActiveRecord::base we can use the find method to find all the pages in the system.

Now extend the index view like so:

```
<h1>Pages</h1>
<% for page in @pages %>
  <div>
    <h2><%= page.title %></h2>
    <p><%= page.introduction %></p>
    <p>
```

```
      <%= link_to 'Show', :action => "show", :id => page.id %> |
      <%= link_to 'Edit', :action => "edit", :id => page.id %> |
      <%= link_to 'Destroy', :action => "destroy", :id => page.id, :confirm =>
'Are you sure?' %>
    </p>
  </div>
  <hr />
<% end %>
<p>
  <%= link_to 'New page', :action => "new" %>
</p>
```

Here we have used a for loop to put the title and introduction text into divs. Notice we've also added links to show, edit and destroy the page. These are the rest of the CRUD actions which we'll be implementing shortly.

```
View localhost:3000/pages.
```

Show Action

The Show action accepts a parameter: the id of the page, and pulls the page object out of the database ready to be displayed. Because Page extends ActiveRecord::Base this is quite easy. Extend the Pages controller like so. This listing has been shortened for brevity:

This time find accepts a parameter. This is taken from a hash called params. The params hash contains any information sent by the user either from a form or in the url. Here it's used to identify a page.

```
class PagesController < ApplicationController

  ...

  def show
    @page = Page.find params[:id]
  end

end
```

If you want to know where params[:id] comes from, look at your routes.

Show View

We'll now need to create a view to render our page. Create a file called /app/views/pages/show.html.erb and insert the following:

```
<h2><%= @page.title %></h2>
<p><%= @page.introduction %></p>
<div class="content">
  <%= @page.text %>
</div>
<p>
  <%= link_to 'Index', :action => "index", :id => @page.id %> |
  <%= link_to 'Edit', :action => "edit", :id => @page.id %> |
  <%= link_to 'Destroy', :action => "destroy", :id => @page.id, :confirm => 'Are
you sure?' %>
</p>
```

Edit and Update Actions

The Edit and Update actions are similar to the New and Create actions except they work on an existing page.

Extend the pages controller like so (listing edited for brevity)

```
class PagesController < ApplicationController

  def edit
    @page = Page.find(params[:id])
  end

  def update
    @page = Page.find(params[:id])
    if @page.update_attributes(params[:page])
      flash[:notice] = 'Page was successfully updated.'
      redirect_to:controller => 'pages'
    else
      render :action => "edit"
    end
  end

end
```

The Edit View

The edit view is similar to the create view. Create a file called /app/views/edit.html.erb containing the following code. This is largely cut and pasted from the new.html.erb file.

```
<h1>Edit page</h1>
<% form_for @page, :url => {:action => 'update', :id => @page} do |f| %>
<%= f.error_messages %>
  <p>
    <%= f.label :title %>
    <br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :introduction %>
    <br />
    <%= f.text_field :introduction %>
  </p>
  <p>
    <%= f.label :text %>
    <br />
    <%= f.text_area :text %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>
```

If we click the edit link next to any of our existing pages we should now be able to edit the page.

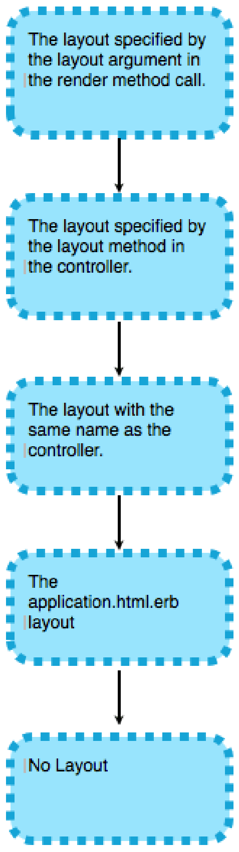
When you click update you should see that the page has changed.

Layouts

Layouts allow us to provide a standard layout for our templates. Layouts live in the views/layouts directory. A precedence hierarchy which determines which one gets used. See the diagram on the next page for details on this.

The Layouts Precedence Hierarchy

Layouts further up the hierarchy are rendered in preference over layouts further down if they are present.


The layout specified by the layout argument in the render method call.
The layout specified by the layout method in the controller.
The layout with the same name as the controller.
The application.html.erb layout
No Layout

Implementing the application.html.erb layout

Here we want to implement a basic layout so we go to the bottom of the hierarchy and create a file called app/views/layouts/application.html.erb containing something like the following:

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Pages: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'style' %>
</head>
<body>
  <h1>My CMS</h1>
  <p style="color: green"><%= flash[:notice] %></p>
  <%= yield  %>
</body>
</html>
```

There are several important things to note here.

The Yield statement yields control to the template. Put this where you want your content to go. The flash[:notice] renders the contents of the flash hash.

Adding a stylesheet

By default stylesheets live in the /public/stylesheets directory. Create one here called style.css and it will be imported. add some code to it to make your application sing!

Validation

As you might expect, validation in Ruby takes place inside our model.

Validating before save

Open up the app/models/page.rb file and insert the following code

```
class Page < ActiveRecord::Base
  validates_presence_of :title
  validates_uniqueness_of :title
end
```

Try to save a page with a blank title. The save method will return false and you'll be shown an error. Brilliantly simple.

Filters

Filters are a great way to reduce typing. One of the central rails axioms is Do not Repeat Yourself (DRY). In the rails community people often talk about drying up an application by reducing unnecessary duplication. Filters are a great way to do this. Using a before filter we can rewrite our pages controller like so:

```
class PagesController < ApplicationController

  before_filter :get_page, :only => [:show, :edit, :update, :delete]

  def index
    @pages = Page.find :all
  end

  def show
  end
  def new
    @page = Page.new
  end
  def create
    @page = Page.new(params[:page])
    if @page.save
      flash[:notice] = 'Page was successfully created.'
      redirect_to:controller => 'pages'
    else
      render :action => "new"
    end
  end

  def edit
  end
  def update
    if @page.update_attributes(params[:page])
      flash[:notice] = 'Page was successfully updated.'
      redirect_to:controller => 'pages'
    else
      render :action => "edit"
    end
  end

  def get_page
    @page = Page.find(params[:id])
  end
```

end

You can define before filters in your app/controllers/application.rb file. This is a great way to share filters between controller. It’s also a great way to implement user login validation.

We can use filters to implement login. Run a before filter before anything you want to secure. Then

Sessions

Rails allows you to store information about a user’s browsing session. It does this by setting a cookie on the user’s machine then associating the value in that cookie with a database entry.

Configuring Sessions

To use sessions we must enable them in our configuration file.

edit the /config/environment.rb file and uncomment the line:

```
# config.action_controller.session_store = :active_record_store
```

This tells rails to store our sessions in the database. In any medium to large data backed application this is where you’ll want them to be.

Setting up the sessions table

Now we need to create the database tables where our sessions will be stored. There’s a shorthand for this

```
rake db:sessions:create
rake db:migrate
```

Using the session hash

We can now write to and read from the session using the session object from our controller or view. We can write anything we like, to give a simple example:

```
session[:admin_user] = true
```

Or if you’re implementing login:

```
session[:user_id] = 123
```

Or just to remember which tab a user was on:

```
session[;active_tab] = "inbox"
```

Flash

Closely related to sessions is the flash hash. Flash is a temporary session. Anything stored in the flash hash will be available to the next controller. We can us it as a simple way to pass data from one controller to the next.

This is useful for things like error messages. For example, if you’re controller encounters an error, it might want to forward to a different controller and show a message.

Session security

There are two ways to store sessions

Serverside:

The session cookie stores a hash which is retrieved and used as a key to pull serialised data from the database.

Clientside:

The session cookie itself holds encrypted serialised data.

Client side cookies are the default because they are faster, and most of what you might want to store in the session is not secret.

You would be unwise to hold anything like :admin_user = false clientside for obvious reasons. Even though the cookie is encrypted, it’s still not a good idea. Likewise you should beware of storing :user_id = 123, as the admin user will commonly have an id of 1.

Instead, if I’m using the default client side cookies I generally store something like:

```
user_hash = fdwhyuklr23470gfetm45ryfo79dGdfyig76GwTRiuypgf5432[uo
```

where user hash is a random length random string, almost impossible to guess.

Exercise

Use your knowledge of filters and sessions to create a very simple session controller. When you hit sessions#new it chceks the params hash for an email and password. If they match it’s default, it sets session[:admin_user] = true.

The session controller should have 3 methods, new, create and destroy. New should render a login form, create should check the user’s name and password against a hardcoded value, and should store :logged_in = true in the session, destroy should set logged_in to be nil.

Modify your application to hide unauthorised links by putting if statements around the links that check the session.

Create a before filter to redirect unauthorised users to the session#new method if they try to edit a page.

Appendix 1: Environment setup

To develop in Ruby you'll need:

A copy of the Ruby interpreter. The most common at time of writing are 1.9.2 or 2.0. Choose the one that runs on your platform.

A text editor, usually Sublime Text 2.

Appendix 2: Further reading

Eigienclass or Singleton Class

http://coreyhoffstein.com/2009/02/24/rubys-eigenclass-and-metaclass/

Rake

http://jasonseifer.com/2010/04/06/rake-tutorial

Controllers

One controller method for each action

Subclass Application Controller