Using the Power of Ruby/JRuby and Rails to Develop Robust Applications

Sang Shin
Technology Architect
javapassion.com
Topics

- Ruby language basics
- Ruby blocks
- Ruby meta-programming
- JRuby basics
- Rails basics
- REST
- Testing
- Ajax
Ruby Language Basics
Topics

• Ruby naming convention
• Ruby types
  > String, Hash, Symbol
• Ruby class
• Inheritance
• Methods
• Variables
Ruby Naming Conventions
Ruby Naming Conventions

• Ruby file - .rb suffix
  > myprog.rb

• Class & Module names – MixedCase
  > MyClass

• methods - lower case with underscores
  > my_own_method

• local variables – lower case with underscores (same as methods)
  > my_own_variable

• Instance variables - @ prefix to variable name
  > @my_instance_variable
Ruby Types
Ruby Types

- String
- Number
- Symbol
- Array
- Hash
Ruby Types:

Strings
String Literals

• One way to create a String is to use single or double quotes inside a Ruby program to create what is called a string literal

```ruby
puts 'Hello world'
puts "Hello world"
```

• Double quotes allow you to embed variables or Ruby code inside of a string literal – this is commonly referred to as interpolation.

```ruby
def my_method(name)
    puts "Your name is #{name}"
end
```
Ruby Types:

Symbols
What is Symbol?

• A Ruby symbol is the internal representation of a name
• It is a class in Ruby language
  :my_value.class #=> Symbol
• You construct the symbol for a name by preceding the name with a colon.
  :my_symbol
• Atomic, immutable and unique
  > Can't be parsed or modified
  > All references to a symbol refer to the same object
  :my_value.equal?(:my_value) #=> true
  "my_value".equal?("my_value") #=> false
Symbols vs. Strings

• Symbols are always interchangeable with strings
  > In any place you use a string in your Ruby code, you can use a symbol

• Important reasons to use a symbol over a string
  > If you are repeating same string many times in your Ruby code, let's say 10000 times, it will take 10000 times of memory space of the string while if you are using a symbol, it will take a space for a single symbol

• Minor reasons to use a symbol over a string
  > Symbol is easier to type than string (no quotes to type)
  > Symbol stands out in the editor
  > The different syntax can distinguish keys from values in hash
    > :name => 'Brian'
Ruby Types: Hash
Hash

• Hashes are basically the same as arrays, except that a hash not only contains values, but also keys pointing to those values.

• Each key can occur only once in a hash.

• A hash object is created by writing `Hash.new` or by writing an optional list of comma-separated `key => value` pairs inside curly braces

```ruby
hash_one   = Hash.new
hash_two   = {}                             # shorthand for Hash.new
hash_three = {"a" => 1, "b" => 2, "c" => 3}
```
Hash and Symbol

- Usually Symbols are used for Hash keys (allows for quicker access), so you will see hashes declared like this:

```ruby
hash_sym = { :a => 1, :b => 2, :c => 3 }
```
Where Do Symbols Typically Used?

- Symbols are often used as
  - Hash keys (`:name => 'Brian', :hobby => 'golf'`)
  - Arguments of a method (`:name, :title`)
  - Method names (`:post_comment`)
- Symbols are used in Rails pervasively
Ruby Class
Ruby Classes

- Every object in Ruby has a class. To find the class of an object, simply call that object's `class` method.

  ```ruby
  "This is a string".class #=> String
  9.class #=> Fixnum
  ["this","is","an","array"].class #=> Array
  {:this => "is", :a => "hash"}.class #=> Hash
  :symbol.class #=> Symbol
  ```
Defining a Class

- Use `class` keyword

```ruby
class Chocolate
  def eat
    puts "That tasted great!"
  end
end
```
Instantiation of an Object

- An object instance is created from a class through the a process called instantiation.
- In Ruby this takes place through a Class method `new`.
  
  ```ruby
  an_object = MyClass.new(parameters)
  ```
- This function sets up the object in memory and then delegates control to the `initialize` function of the class if it is present. Parameters passed to the new function are passed into the `initialize` function.
  
  ```ruby
  class MyClass
    def initialize(parameters)
    end
  end
  ```
Class Example

• Simple RocketShip Class

```ruby
class RocketShip < Object
  attr_accessor :destination

  def initialize(destination)
    @destination = destination
  end

  def launch()
    "3, 2, 1 Blast off!"
  end
end
```
Class Example

• Single Inheritance

class RocketShip < Object  # < Object is optional like in Java
  attr_accessor :destination

  def initialize(destination)
    @destination = destination
  end

  def launch()
    "3, 2, 1 Blast off!"
  end
end
Class Example

• Constructors in Ruby are named initialize

```ruby
class RocketShip < Object
  attr_accessor :destination

  def initialize(destination)
    @destination = destination
  end

  def launch()
    "3, 2, 1 Blast off!"
  end

end
```

# new() allocates a RocketShip instance and initialize()
# initializes that instance
r = RocketShip.new('Netptune')
Class Example

- Attributes are easily defined

```ruby
class RocketShip < Object
  # No need to define getter and setter for an attribute
  attr_accessor :destination

  def initialize(destination)
    @destination = destination
  end

  def launch()
    "3, 2, 1 Blast off!"
  end
end

r = RocketShip.new
r.destination = 'Saturn'
```
Ruby Class: Inheritance
Inheritance

• A class can inherit functionality and variables from a superclass, sometimes referred to as a parent class or base class. (Same in Java)

• Ruby does not support multiple inheritance and so a class in Ruby can have only one superclass. (Same in Java)

• All non-private variables and functions are inherited by the child class from the superclass. (Same in Java)
Overriding a method

- If your class overrides a method from parent class (superclass), you still can access the parent's method by using 'super' keyword

```ruby
class ParentClass
  def a_method
    puts 'b'
  end
end

class ChildClass < ParentClass
  def a_method
    super           # Call a_method of a parent class
    puts 'a'
  end
end

instance = ChildClass.new
instance.a_method
```
Methods
Method Definitions

• Methods are defined using the keyword `def` followed by the method name.

• By convention method names that consist of multiple words have each word separated by an underscore.

```ruby
def output_something(value)
  puts value
end
```
Class Method vs. Instance Method

- A class can contain both class and instance methods
- Class method is defined with `self.<method_name>`

```ruby
class MyClass
  def self.find_everybody
    find(:all)
  end
  def my_instance_method
  end
end
```

- Class method is invoked with a class

```ruby
MyClass.find_everybody
```
How to Invoke Methods

• Methods are called using the following syntax:
  \texttt{method\_name(parameter1, parameter2,\ldots)}

• The parentheses can be omitted, however
  \texttt{method\_name parameter1, parameter2}  \# with parameters
  \texttt{method\_name}  \# with no parameters

• If you use method result immediately, then you have to use parentheses:
  \# You need to use parentheses if you want to work with the result immediately.
  \# eg. If a method returns an array and we want to reverse element order:
  \texttt{results = method\_name(parameter1, parameter2).reverse}
Return Values

- Methods return the value of the last statement executed. The following code returns the value x+y.

```python
def calculate_value(x, y):
    x + y  # return statement is optional
end
```

- An explicit return statement can also be used to return from function with a value, prior to the end of the function declaration. This is useful when you want to terminate a loop or return from a function as the result of a conditional expression.
Methods:
Arguments
Default Value Argument

• A default parameter value can be specified during method definition to replace the value of a parameter if it is not passed into the method or the parameter's value is nil.

```ruby
def some_method(value='default', arr=[])  
  puts value  
  puts arr.length
end

some_method('something')
```

• The method call above will output:

```
something
0
```
Variable Length Argument List

• The last parameter of a method may be preceded by an asterisk (*), which is sometimes called the 'splat' operator. This indicates that more parameters may be passed to the function. Those parameters are collected up and an array is created.

```ruby
def calculate_value(x, y, *otherValues)
  puts otherValues
end
```

```ruby
calculate_value(1, 2, 'a', 'b', 'c')
```

• In the example above the output would be `['a', 'b', 'c']`. 
Hash Argument

• Another technique that Ruby allows is to pass a Hash argument when invoking a function, and that gives you best of all worlds: named parameters, and variable argument length.

```ruby
def accepts_hash( var )
  print "got: ", var.inspect    # will print out what it received
end

# Pass a hash as an argument
accepts_hash( {:arg1 => 'giving arg1', :argN => 'giving argN'} )
# => got: {:argN=>"giving argN", :arg1=>"giving arg1"}
```
Parentheses ( ) for the Arguments, Braces { } for a Hash Argument

• Parentheses can be omitted for the arguments
• If the last argument is a Hash, braces { } of the Hash can be omitted. The following three work the same.

```ruby
# Arguments are with enclosed ( ), hash is enclosed with braces {}
accepts_hash( { :arg1 => 'giving arg1', :argN => 'giving argN' } )

# No parentheses for arguments, {} for a hash
accepts_hash {:arg1 => 'giving arg1', :argN => 'giving argN'}

# No parentheses for arguments, no {} for a hash
accepts_hash :arg1 => 'giving arg1', :argN => 'giving argN'
```
Calling a Method with a Code Block

- If you are going to pass a code block to a function, you need parentheses for arguments.

```ruby
# You need parentheses for arguments since there is a block
accepts_hash({ :arg1 => 'giving arg1', :argN => 'giving argN' }) { |s| puts s }
accepts_hash({ :arg1 => 'giving arg1', :argN => 'giving argN' }) { |s| puts s }

# Compile error
accepts_hash :arg1 => 'giving arg1', :argN => 'giving argN' { |s| puts s }
```
Ruby Blocks
Topics

- What is a block?
- How does a block look like?
- How does a block get passed and executed?
What is a Block?
What is a block?

- Blocks are basically nameless functions.
  > You can think of it as a nameless chunk of code.
- You can pass a nameless function to another function (I will call it a target function in this presentation), and then that target function can invoke the passed-in nameless function.
  > For example, a target function could perform iteration by passing one item at a time to the nameless function.

If you don't understand what is being said here, don't worry about it. We will explore this with examples!
How Does a Block Look Like?
How to Represent a Block?

• A block can be represented in two different formats - these two formats are functionally equivalent
• Convention is use \{ \} for a single line block to use do ... end for multi-line block

```ruby
puts "----First format of code block containing code fragment between { and }"
[1, 2, 3].each { puts "Life is good!" }
```

```ruby
puts "----Second format of code block containing code fragment between do and end"
[1, 2, 3].each do
  puts "Life is good!"
end
```
How Does a Block Get Passed & Executed?
How a block is passed & executed

- When a method is invoked, a block can be passed (attached)
- The yield() method in the invoked method (target function) executes the passed block

```ruby
puts "----Define MyClass which invokes yield"
class MyClass
  def command()
    # yield will execute the attached block to the method
    yield()
  end
end

puts "----Create object instance of MyClass"
m = MyClass.new
puts "----Call command method of the MyClass passing a block"
m.command {puts "Hello World!'"}
```
How a block receive arguments

- A block can receive arguments - they are represented as comma-separated list at the beginning of the block, enclosed in pipe characters:

```
puts "----Define MyClass which invokes yield"
class MyClass
  def command1()
    # yield will execute the supplied block
    yield(Time.now)
  end
end

puts "----Create an object instance of MyClass"
m = MyClass.new

puts "----Call command1 method of the MyClass"
m.command1() {|x| puts "Current time is #{x}"}
```
Example: Block Receive Argument

- *each* method of *Array* class pass an element as an argument

```ruby
puts "----First format of code block containing code fragment between { and }"
[1, 2, 3].each { |n| puts "Number #{n}" }

puts "----Second format of code block containing code fragment between do and end"
[1, 2, 3].each do |n|
  puts "Number #{n}"
end
```
How a block can receive arguments

• A block can receive multiple arguments

```ruby
puts "---Define a method called testyield"
def testyield
  yield(1000, "Sang Shin")
  yield("Current time is", Time.now)
end

puts "----Call testyield method"
testyield { |arg1, arg2| puts "#{arg1} #{arg2}" }

• Result

---Define a method called testyield
----Call testyield method
1000 Sang Shin
Current time is Mon Jun 30 09:14:56 -0400 2008
Ruby
Meta-Programming
Topics

• What is and Why Meta-programming?
• Ruby language characteristics (that make it a great meta-programming language)
• Introspection
• Object#send
• missing_method
What is Meta-Programming?
What is Meta-Programming?

- Meta-programming is the writing of computer programs that write or manipulate other programs (or themselves) as their data.
Why Meta-Programming?

• Provides higher-level abstraction of logic
  > Easier to write code
  > Easier to read code

• Meta-programming feature of Ruby language is what makes Rails a killer application.
  > For example, the Rails declarations such as "find_by_name", "belongs_to", “has_many” are possible because of the Meta-programming feature of Ruby language.
Classes Are Open

- Unlike Java and C++, in Ruby, methods and instance variables can be added to a class (including core classes provided by Ruby such as \texttt{String} and \texttt{Fixnum}) during runtime.
- Example: Define a new method called \texttt{encrypt} for the String class

```ruby
class String
  def encrypt
    tr "a-z","b-za"
  end
end
```

```ruby
puts "cat"
puts "cat".encrypt
```
Classes Are Open

• Benefits
  > Applications can be written in higher level abstraction
  > More readable code
  > Less coding

• How it is used in Rails
  > One can open up Rails classes and add new features to them.
  > Rails integration testing is a good example
Every Method Call Has a Receiver

- Default receiver is `self`
Introspection
What is Introspection?

• Being able to find information on an object during runtime

• Examples
  > Object#respond_to?
  > Object#class
  > Object#methods
  > Object#class.superclass
  > Object#class.ancestors
  > Object#private_instance_methods()
  > Object#public_instance_methods()
  > ...

Dynamic Method Invocation through Object#send
Dynamic Method Invocation in Ruby

• In Ruby, an object’s methods are not fixed at any compilation time but can be dynamically extended or modified at any point.

• Calling a method directly by name is allowed as expected
  > an_object_instance.hello(“Good morning!”)

• It is also possible to invoke generically any object method by using a string or symbol variable to specify the target method
  > an_object_instance.send(“#{name_of_method}”, args)
  > an_object_instance.send(:my_method, args)
obj.send(symbol [, args...])

- Invokes the method identified by symbol, passing it any arguments specified.

    class Klass
      def hello(*args)
        "Hello " + args.join(' ')
      end
    end

    k = Klass.new

    # The following statements are equivalent
    puts k.hello("gentle", "readers")  #=> "Hello gentle readers"
    puts k.hello "gentle", "readers"   #=> "Hello gentle readers"
    puts k.send(:hello, "gentle", "readers") #=> "Hello gentle readers"
    puts k.send :hello, "gentle", "readers" #=> "Hello gentle readers"
missing_method
NoMethodError Exception

• If a method that is not existent is in a class is invoked, *NoMethodError* exception will be generated

```ruby
class Dummy
end

puts "----Call a method that does not exist in the Dummy class and expect NoMethodError exception."
dummy = Dummy.new
dummy.call_a_method_that_does_not_exist
```
method_missing Method

• If `method_missing(m, *args)` method is defined in a class, it will be called (instead of `NoMethodError` exception being generated) when a method that does not exist is invoked

```ruby
class Dummy
  def method_missing(m, *args)
    puts "There's no method called #{m} here -- so method_missing method is called."
    puts "   with arguments #{args}"  
  end
end

dummy = Dummy.new
dummy.a_method_that_does_not_exist
```
How method_missing Method is used in Rails

- Rails' `find_by_xxxx()` finder method is implemented through `method_missing`.

```ruby
class Finder
  def find(name)
    # Rails (actually ActiveRecord) constructs a find() method with correct set of parameters
    puts "find(#{name}) is called"
  end

  def method_missing(name, *args)
    if /^find_(.*)=~/ =~ name.to_s
      return find($1)
    end
    super
  end
end

f = Finder.new
f.find("Something")
f.find_by_last_name("Shin")
f.find_by_title("Technology Architect")
```
Rails Basics
Topics

• What is and Why Ruby on Rails?
• Building HelloWorld Rails application step by step
• Key concepts of Rails application development
  > App directory structure (MVC)
  > Environment
  > Rake
  > Generator
  > Migration
  > Rails console
What is and Why Ruby on Rails (RoR)?
What Is “Ruby on Rails”?

• A full-stack MVC web development framework
• Written in Ruby
  > Rails leverages various characteristics of Ruby language -
    meta-programming, closure, etc.
• First released in 2004 by David Heinemeier Hansson
• Gaining popularity
“Ruby on Rails” MVC

source: http://www.ilug-cal.org
“Ruby on Rails” Principles

• Convention over configuration
  > Why punish the common cases?
  > Encourages standard practices
  > Everything simpler and smaller

• Don’t Repeat Yourself (DRY)
  > Framework written around minimizing repetition
  > Repetitive code harmful to adaptability

• Agile development environment
  > No recompile, deploy, restart cycles
  > Simple tools to generate code quickly
  > Testing built into the framework
Building “Hello World” Rails Application Step by Step
Steps to Follow

1. Create “Ruby on Rails” project
   > Rails generates directory structure
2. Create Database (using Rake)
3. Create Models (using Generator)
4. Create Database Tables (using Migration)
5. Create Controllers (using Generator)
6. Create Views
7. Set URL Routing
   > Map URL to controller and action
Demo:
Building “Hello World” Rails Application Step by Step.
1. Create “Ruby on Rails” Project
1. Create “Ruby on Rails” Project

The directory structure along with boilerplate files of the application is created:

```
# MySQL. Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
#    gem install mysql
# On Mac OS X:
#    sudo gem install mysql --with-mysql-dir=/usr
# On Mac OS X Leopard:
#    sudo env ARCHFLAGS="-arch i386" gem install mysql
#    This sets the ARCHFLAGS environment variable
# On Windows:
#    gem install mysql
#    Choose the win32 build.
#    Install MySQL and put its /bin directory on PATH.
# And be sure to use new-style password hashing:
#    http://dev.mysql.com/doc/refman/5.0/en/old-cli

development:
  adapter: mysql
  encoding: utf8
  database: helloworld_development
  username: root
  password: helloworld
```

```
host: localhost
```

```
# Warning: The database defined as 'test' will be re-generated from your development database when
# Do not set this db to the same as development or
```
Directory Structure of a Rails Application

• When you ask NetBeans to create a Rails project - internally NetBeans uses the rails' helper script -, it creates the entire directory structure for your application.
  > The boiler plate files are also created
  > The names of the directories and files are the same for all Rails projects
• Rails knows where to find things it needs within this structure, so you don't have to tell it explicitly.
Directory Structure of a Rails Application

- **app**: Holds all the code that's specific to this particular application.
  - **app/controllers**: Holds controllers that should be named like `hello_controller.rb` for automated URL mapping. All controllers should descend from ` ApplicationController` which itself descends from ` ActionController::Base`. 
  - **app/models**: Holds models that should be named like `message.rb`. Most models will descend from ` ActiveRecord::Base`. 
  - **app/views**: Holds the template files for the view that should be named like `hello/say_hello.rhtml` for the `HelloController#say_hello` action.
Directory Structure of a Rails Application

- **app**
  - **app/views/layouts**: Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views. In your views, define a layout using the `<tt>layout :default</tt>` and create a file named `default.rhtml`. Inside `default.rhtml`, call `<% yield %>` to render the view using this layout.
  - **app/helpers**: Holds view helpers that should be named like `hello_helper.rb`. These are generated for you automatically when using `script/generate` (Generator) for controllers. Helpers can be used to wrap functionality for your views into methods.
Directory Structure of a Rails Application

- **config**: Holds configuration files for the Rails environment, the routing map, the database, and other dependencies.
  
  > config/environments
  > config/initializers
  > boot.rb
  > database.yml
  > environment.rb
  > routes.rb
Learning Point: Environments
What is an Environment?

- Rails provides the concept of environments - *development, test, production*
- As a default, different database is going to be used for different environment.
  > Therefore each environment has its own database connection settings.
- It is easy to add custom environments
  > For example, staging server environment
- Rails always runs in only one environment
  > Dictated by ENV['RAILS_ENV'] (same as RAILS_ENV)
config/database.yml

development:
  adapter: mysql
  encoding: utf8
  database: helloname_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  encoding: utf8
  database: helloname_test
  username: root
  password:
  host: localhost
2. Create Database using “Rake”
Creating Database

- Creating and dropping of databases are done using "Rake"
Creating Database

- After create rake task is performed, `<project_name>_development` database, for example, `helloworld_development` is created
Learning Point: What is Rake?
What is “Rake”?

• Rake is a build language for Ruby.
• Rails uses Rake to automate several tasks such as
  > creating and dropping databases
  > running tests
  > updating Rails support files.
• Rake lets you define a dependency tree of tasks to be executed
How does “Rake” Work?

- Rake tasks are loaded from the file `Rakefile`
- Rails rake tasks are under `<project-name>/lib/tasks`
- You can put your custom tasks under `lib/tasks`
3. Create a Model through “Generator”
What is a Model?

• In the context of MVC pattern, a Model represents domain objects such as message, school, product, etc.

• A model has attributes and methods.
  > The attributes represents the characteristics of the domain object, for example, a message model might have length, creator as attributes.
  > The methods in a model contains some business logic.

• Most models have corresponding database tables. For example, a message model will have messages table.

• Most model classes are ActiveRecord type
Creating a Model using Generator
Creating a Model using Generator

**Rails Generator**

- **Generate:** model
- **Arguments:** Message greeting:string

**Description:**
- Stubs out a new model. Pass the model name, either CamelCased or under_scored, and an optional list of attribute pairs as arguments.
- Attribute pairs are column_name:sql_type arguments specifying the model's attributes. Timestamps are added by default, so you don’t have to specify them by hand as 'created_at:datetime updated_at:datetime'.
- You don’t have to think up every attribute up front, but it helps to sketch out a few so you can start working with the model immediately.
- This generates a model class in app/models, a unit test in test/unit, a test fixture in test/fixtures/singular_name.yml, and a migration in.
Files That Are Created

• app/models/message.rb (Model file)
  > Models/message.rb in logical view
  > A file that holds the methods for the Message model.

• test/unit/message_test.rb
  > Unit Tests/message_test.rb in logical view
  > A unit test for checking the Message model.

• test/fixtures/messages.yml
  > Test Fixtures/messages.yml in logical view
  > A test fixture for populating the model.

• db/migrate/migrate/001_create_messages.rb
  > Database Migrations/migrate/001_create_messages.rb in logical view
  > A migration file for defining the initial structure of the database.
Model Class Example

- *Message* mode in *messages.rb* file

```ruby
class Message < ActiveRecord::Base
end
```
Learning Point:
What is Generator?
What is “Generator”? 

- You can often avoid writing boilerplate code by using the built-in generator scripts of Rails to create it for you. 
  > This leaves you with more time to concentrate on the code that really matters--your business logic.
Learning Point: What is Rails Console?
What is Rails Console?

• The Rails console gives you access to your Rails Environment, for example, you can interact with the domain models of your application as if the application is actually running.
  > Things you can do include performing find operations or creating a new active record object and then saving it to the database.

• A great tool for impromptu testing

• NetBeans runs a script to start Rails Console
Learning Point:
What is Rails Script?
Script

- NetBeans runs Rails Script internally
  > You can run the Script at the commandline

- Useful scripts
  > console
  > generate
  > plugin
  > server
4. Create Database Tables using Migration
Create Database Table using Migration

• You are going to create a database table (in a previously created database) through migration
  > You also use migration for any change you are going to make in the schema - adding a new column, for example

• When you create a Model, the first version of the migration file is automatically created
  > db/migrate/migrate/<timestamp>_create_messages.rb, which defines initial structure of the table

```ruby
class CreateMessages < ActiveRecord::Migration
  def self.up
    create_table :messages do |t|
      t.string :greeting
      t.timestamps
    end
  end
end
```
Performing Migration

```
class CreateMessages < ActiveRecord::Migration
  def self.up
    create_table :messages do |t|
      t.string :greeting
    end
  end

  def self.down
    drop_table :messages
  end
end
```
learning point: What is Migration?
Issues with Schema Changes

- Database schema's keep changing (along with applications that use them)
  > Example: You need to add “email” column to the “customer” table

- Issues with schema changes
  > How do you version control schema changes?
  > How do you go back to previous version of the schema?
  > How do people work with different versions of the schema?
  > How do you convey schema changes to other developers and the production server?
Migration To the Rescue

• Migration can manage the evolution of a schema
• With migrations, you can describe schema changes in self-contained Ruby classes - migration files
• You can check these migration files into a version control system
• Migration files are part of an application structure
• You (and others) can choose a schema version of choice, for example, several versions back from the current one
5. Create a Controller
What is a Controller?

• Action Controllers handle incoming Web requests
• A controller is made up of one or more actions
• Actions are executed to handle the incoming requests and then either render a template or redirect to another action.
• An action is defined as a public method of a controller
• Mapping between a request's URL and an action is specified in the Rails routing map (configuration/routes.rb)
Create a Controller using Generator

- You are going to create a controller using Generator
Example: HelloController

- Controller contains actions, which are defined with `def`

```ruby
class HelloController < ApplicationController
  def say_hello
    @hello = Message.new(:greeting => "Hello World!")
  end
end
```
6. Write a View
What is a View?

• View is represented by a set of templates that get displayed.
• Templates share data with controllers through mutually accessible variables.
• A template can be either in the form of *.rhtml or *.erb file.
  > The *.erb file is searched first by Rails. If there is no *.erb file, then *.rhtml file is used.
Creating *.rhtml file (or *.erb file)

- *.rhtml or *.erb file is created under the directory of /app/views/<controller>
Example *.rhtml file

• say_hello.rhtml

  My greeting message is <%= @hello.greeting %>
  <br/>
  The current time is <%= Time.now %>
7. Set URL Routing
URL Routing

• The Rails routing facility is pure Ruby code that even allows you to use regular expressions.

> Because Rails does not use the web server's URL mapping, your custom URL mapping will work the same on every web server.

> `configuration/routes.rb` file contains the routing setting
routes.rb

ActionController::Routing::Routes.draw do |map|

  map.root :controller => "hello"

  # Install the default routes as the lowest priority.
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
REST Support in Rails
Topics

• What is REST?
• REST support in Rails
• Format of the response via response_to
What is REST?
REST

- **RE**presentational **S**tate **T**ransfer
- Name coined by Roy Fielding in his Ph.D thesis*
- Architectural Style of the Web

* http://ics.uci.edu/~fielding/pubs/dissertation/top.htm
Characteristics of REST

- RESTful services are stateless
  > Each request from client to server must contain all the information necessary to understand the request

- RESTful services have a uniform interface
  > GET, POST, PUT, and DELETE.

- REST-based architectures are built from resources (pieces of information) that are uniquely identified by URIs
  > In a RESTful purchasing system, each purchase order has a unique URI
Characteristics of REST

• In REST system, resources are manipulated through the exchange of representations of the resources
  > For example, a purchase order resource is represented by an XML document.
  > Within a RESTful purchasing system, a purchase order might be updated by using an XML document containing the changed purchase order to its URI

• REST-based architectures communicate primarily through the transfer of representations of resources
  > State is maintained within a resource representation
REST Support in Rails
Rails Support in Rails

- Rails 1.2 adds REST support through routes facility called `map.resources`
- `map.resources` are defined in `routes.rb`
  ```ruby
  ActionController::Routing::Routes.draw do |map|
    map.resources :users
  end
  ...
  end
  ```
- The `map.resources` adds seven new routes and four new route helpers
- Seven actions are defined in the controller
- Scaffolding on a model creates codes for RESTful operations
# Actions Generated

<table>
<thead>
<tr>
<th>Method</th>
<th>URL path</th>
<th>Action</th>
<th>Helper</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong></td>
<td>/users</td>
<td>index</td>
<td>users_url</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>/users</td>
<td>create</td>
<td>users_url</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/users/new</td>
<td>new</td>
<td>new_user_url</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/users/1</td>
<td>show</td>
<td>user_url(:id=&gt;1)</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td>/users/1</td>
<td>update</td>
<td>user_url(:id=&gt;1)</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>/users/1;edit</td>
<td>edit</td>
<td>edit_user_url(:id=&gt;1)</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>/users/1</td>
<td>destroy</td>
<td>user_url(:id=&gt;1)</td>
</tr>
</tbody>
</table>
Actions Generated

- **index** - return a list of resources
- **show** - return the resource identified by the params[:id]
- **new** - construct a new resource, not saved into the table
- **edit** - return the resource identified by the params[:id] in a form suitable for editing
- **create** - create a new resource, saved into the table
- **update** - update the resource identified by the params[:id] with the data associated with the request
- **destroy** - destroy the resource identified by the params[:id]
index action

- **index** action return a list of resources

```ruby
# GET /users
# GET /users.xml
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @users }
  end
end
```
show action

- **show** returns the resource identified by the params[:id] in `show.html.erb`

```ruby
# GET /users/1
# GET /users/1.xml
def show
  @user = User.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @user }
  end
end
```
new action

- **new** - construct a new resource, not saved into the table

```ruby
# GET /users/new
# GET /users/new.xml

def new
  @user = User.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml  { render :xml => @user }
  end
end
```
edit action

- `edit` - return the resource identified by the params[:id] in a form suitable for editing

```ruby
# GET /users/1/edit
def edit
  @user = Post.find(params[:id])
end
```
create action

- **create** - create a new resource, then save it into the table

```ruby
# POST /users
# POST /users.xml

def create
  @user = User.new(params[:user])

  respond_to do |format|
    if @user.save
      flash[:notice] = 'User was successfully created.'
      format.html { redirect_to(@user) }
      format.xml  { render :xml => @user, :status => :created, :location => @user }
    else
      format.html { render :action => "new" }
      format.xml  { render :xml => @user.errors, :status => :unprocessable_entity }
    end
  end
end
```
Format of the response via respond_to
respond_to Block

- From Rails 1.2, within a `respond_to` block, you can select the format of the response depending on what is being requested by the user:
  - HTML (default) - typically for humans
  - XML
  - JSON
  - YAML
How Does a User Request a Format?

- The format request is specified in the routes.rb

```ruby
ActionController::Routing::Routes.draw do |map|
  ...
  # Install the default routes as the lowest priority.
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.format'
end
```
How Does a User Request a Format?

- Specify the formats to support via `format.<format-to-support> { .. }` inside `respond_to` block

```ruby
class WeblogController < ActionController::Base
  def index
    @users = Post.find :all
    respond_to do |format|
      format.html
      format.xml { render :xml => @users.to_xml }
      format.rss { render :action => "feed.rxml" }  
    end
  end
end
```
How Does a User Request a Format?

- Then Rails framework selects the respond format according to the format request of the URL

  GET /weblog       # returns HTML from browser Accept header
  GET /weblog.xml  # returns the XML
  GET /weblog.rss  # returns the RSS
Using the Power of JRuby and Rails to Develop Robust Applications

Sang Shin
Technology Architect
javapassion.com