CO-PRESENTED BY O'REILLY Ruby Central

A Code Walk-Through of the Source Behind `respond_to`

Andrea O. K. Wright, Chariot Solutions
aok@chariotsolutions.com

This talk is about the respond_to mechanism, which makes it easy to support multiple return types for the same action. I will cover how to use respond_to, but the focus of this talk is how respond_to works.

One Action, Multiple Response Formats

```
def show
  @train_station = TrainStation.find(params[:id])
  respond_to do |format|
    format.html
    format.xml { render :xml => @train_station.to_xml }
  end
end
```

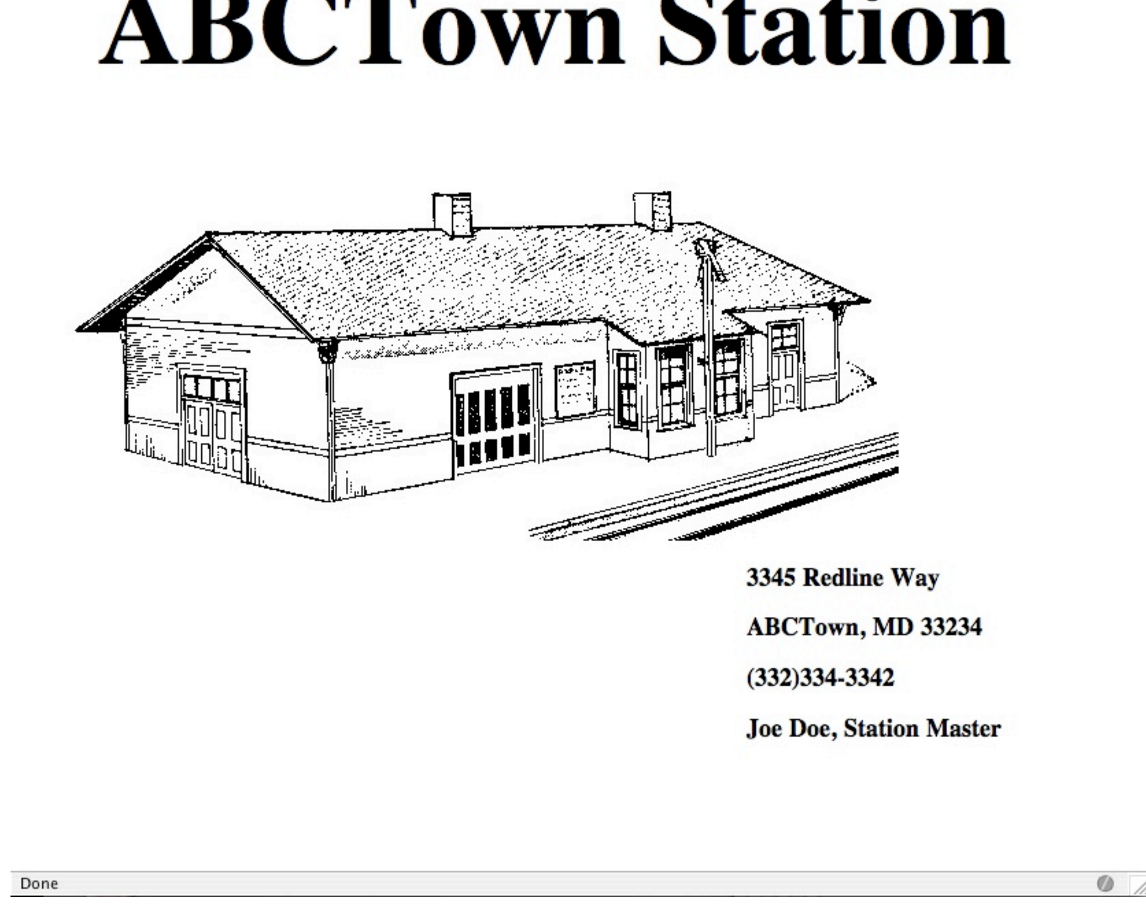

2

This is a basic Rails Controller “show” action. It provides details about a train station. What the part in red -- the respond_to block -- indicates is that....

One Action, Multiple Response Formats

```
def show
  @train_station = TrainStation.find(params[:id])
  respond_to do |format|
    format.html
    format.xml { render :xml => @train_station.to_xml }
  end
end
```

If the client wants HTML, respond with HTML.



Station Drawing from http://www.rrhistorical.com

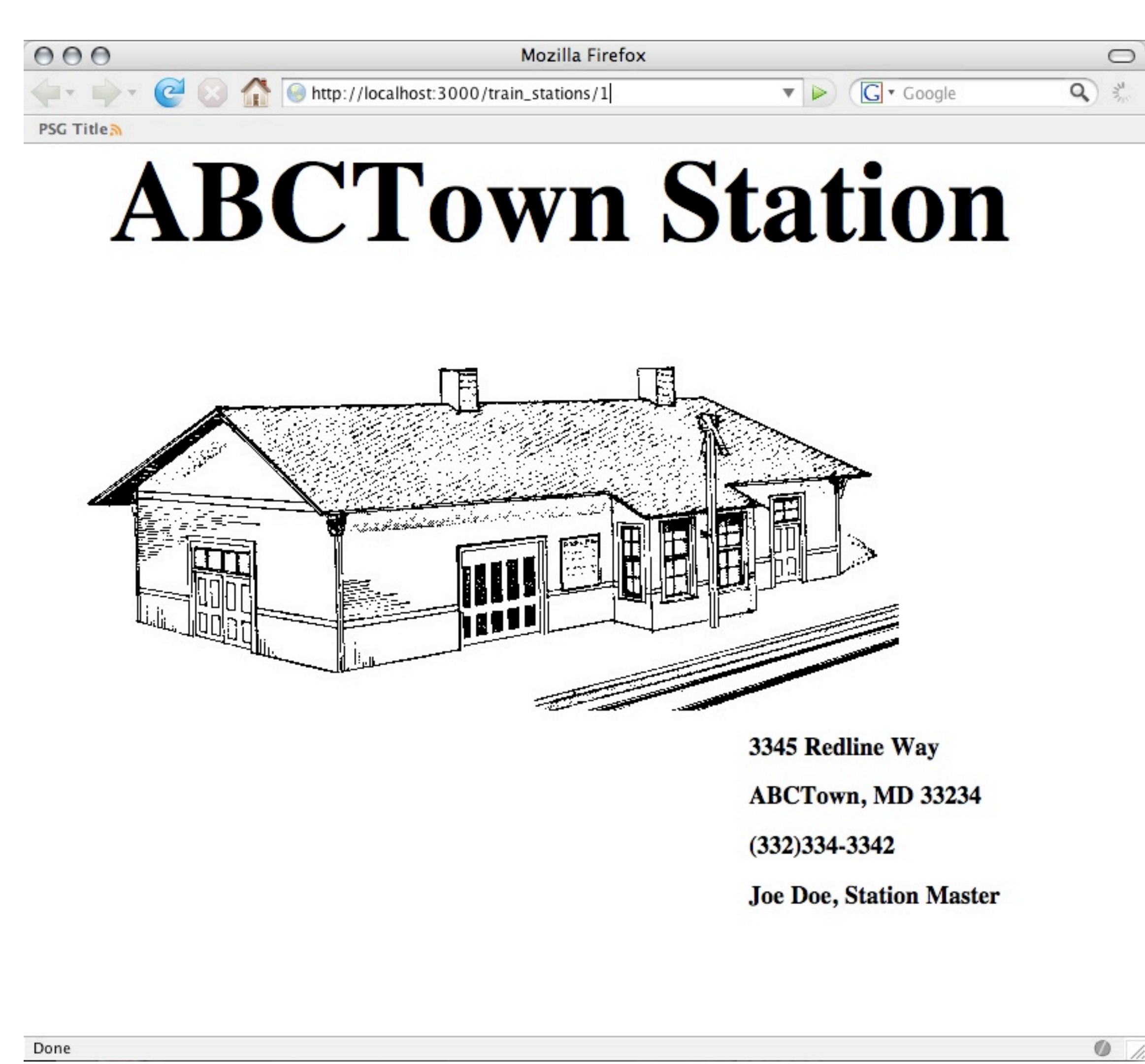
....if the response format the client wants is HTML -- the train station details, like the address and phone number, should be rendered on a Web page. Here the URL mapped to the show action was entered while browsing with Firefox.

And if the response format the client want is XML...

One Action, Multiple Response Formats

```
def show
  @train_station = TrainStation.find(params[:id])
  respond_to do |format|
    format.html
    format.xml { render :xml => @train_station.to_xml }
  end
end
```

If the client wants HTML, respond with HTML.



If the client wants XML, respond with XML.

```
irb> Net::HTTP.start('localhost',3000) do |http|
irb* puts http.get('/train_stations/1/',
    'Accept'=>'application/xml').body
irb> end

<?xml version="1.0" encoding="UTF-8"?>
<train-station>
  <city>ABCTown</city>
  <id type="integer">1</id>
  <name>ABC Station</name>
  <state>MD</state>
  <station-master>Joe Doe</station-master>
  <street>3345 Redline Way</street>
   <zip>33234</zip>
   <phone>(332)334-3342</phone>
</train-station>
```

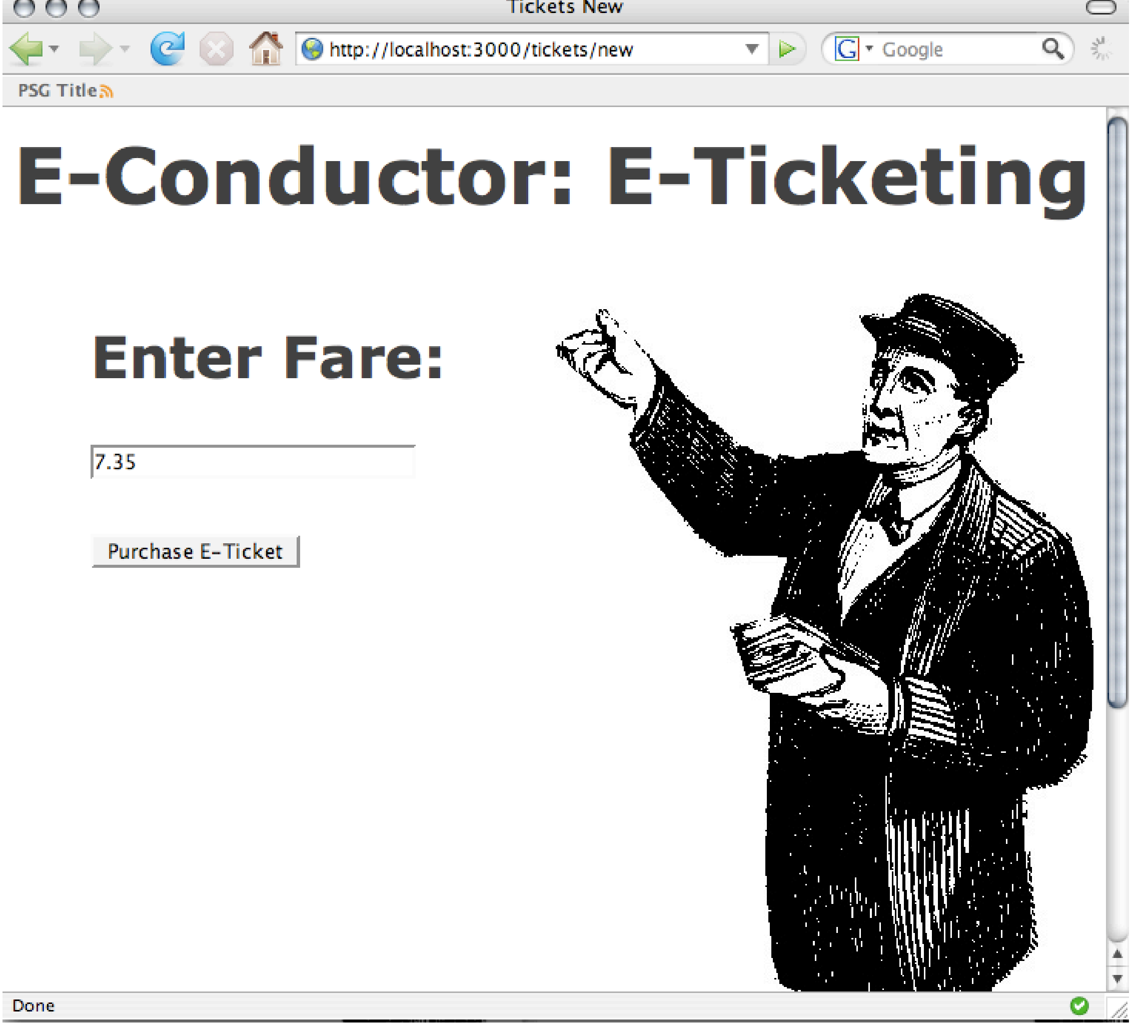
Station Drawing from http://www.rrhistorical.com

...an XML representation of the train station should be provided. Here the request was issued in an interactive Ruby session, an irb session, using the Ruby Net::HTTP library. The Accept header is programatically set to be “application\xml”. The response shown here is the output from calling to_xml on a TrainStation instance.

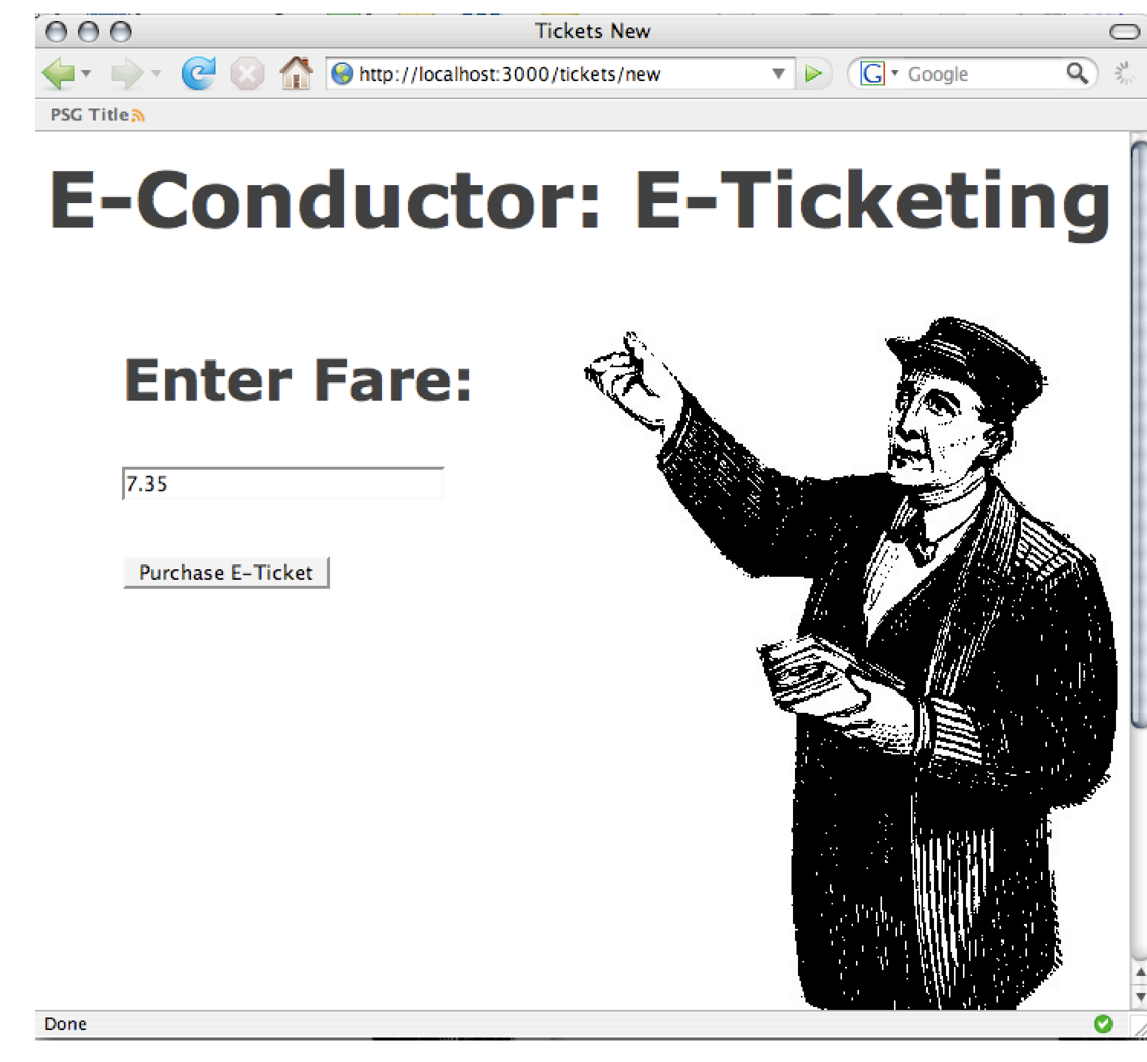
Graceful Degradation with `respond_to`

```
def create
  @ticket = Ticket.new(params)
  ticket.save
  respond_to do |format|
    format.html {redirect_to ticket_url(@ticket)}
    format.js
  end
end
```

JavaScript



HTML



Conductor drawing from http://www.rrhistorical.com

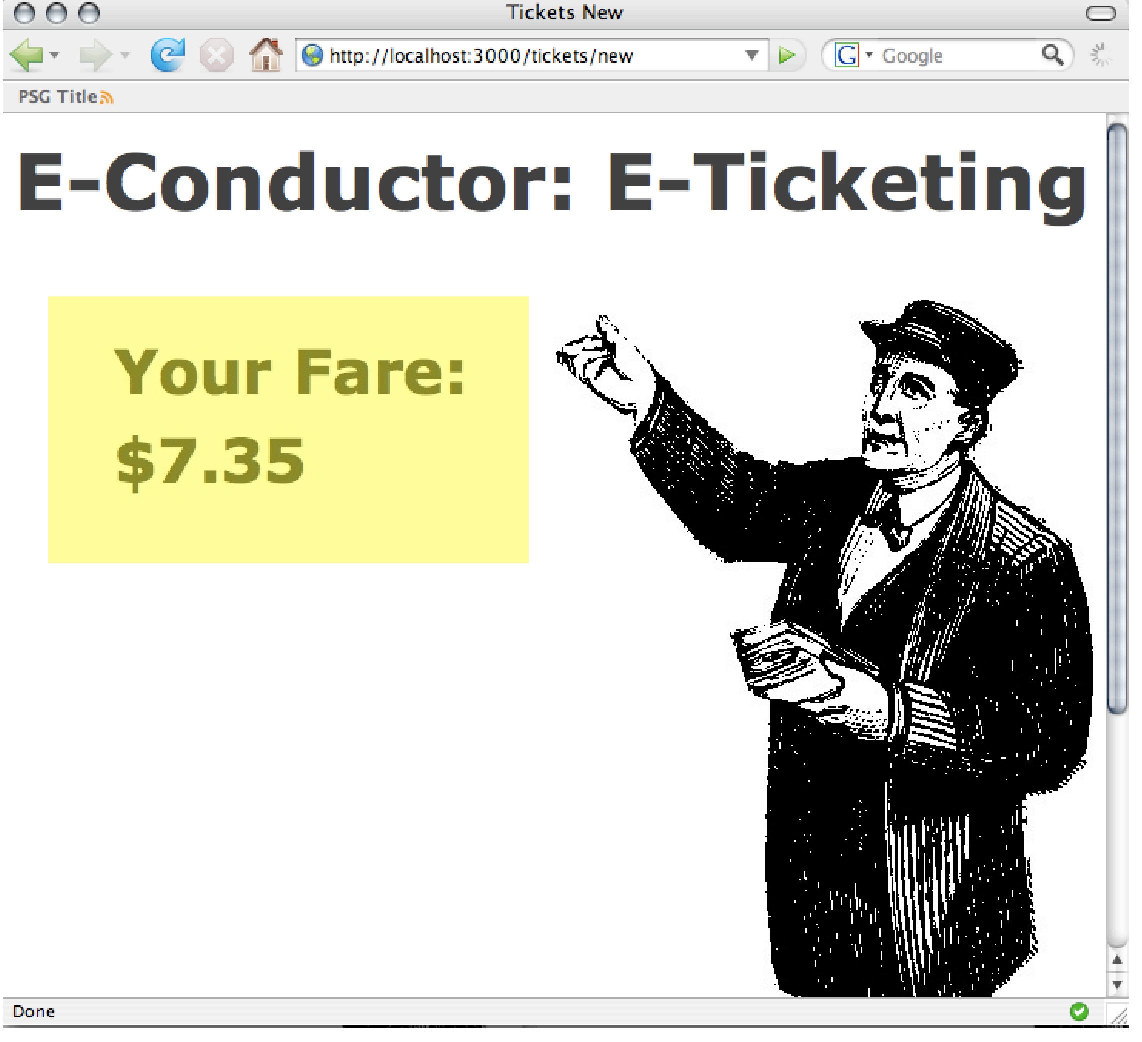
The respond_to mechanism can also play a part in ensuring that an Ajax-enabled web app will degrade gracefully when accessed with a browser that does not support Javascript or if the browser is set to block JavaScript. Here's a simple ticketing page: you type in the ticket price and hit the submit button.

If the browser is open to Javascipt, the create.rjs template is invoked. It includes JavaScript to replace the input field with plain text that represents the ticket value and then ...

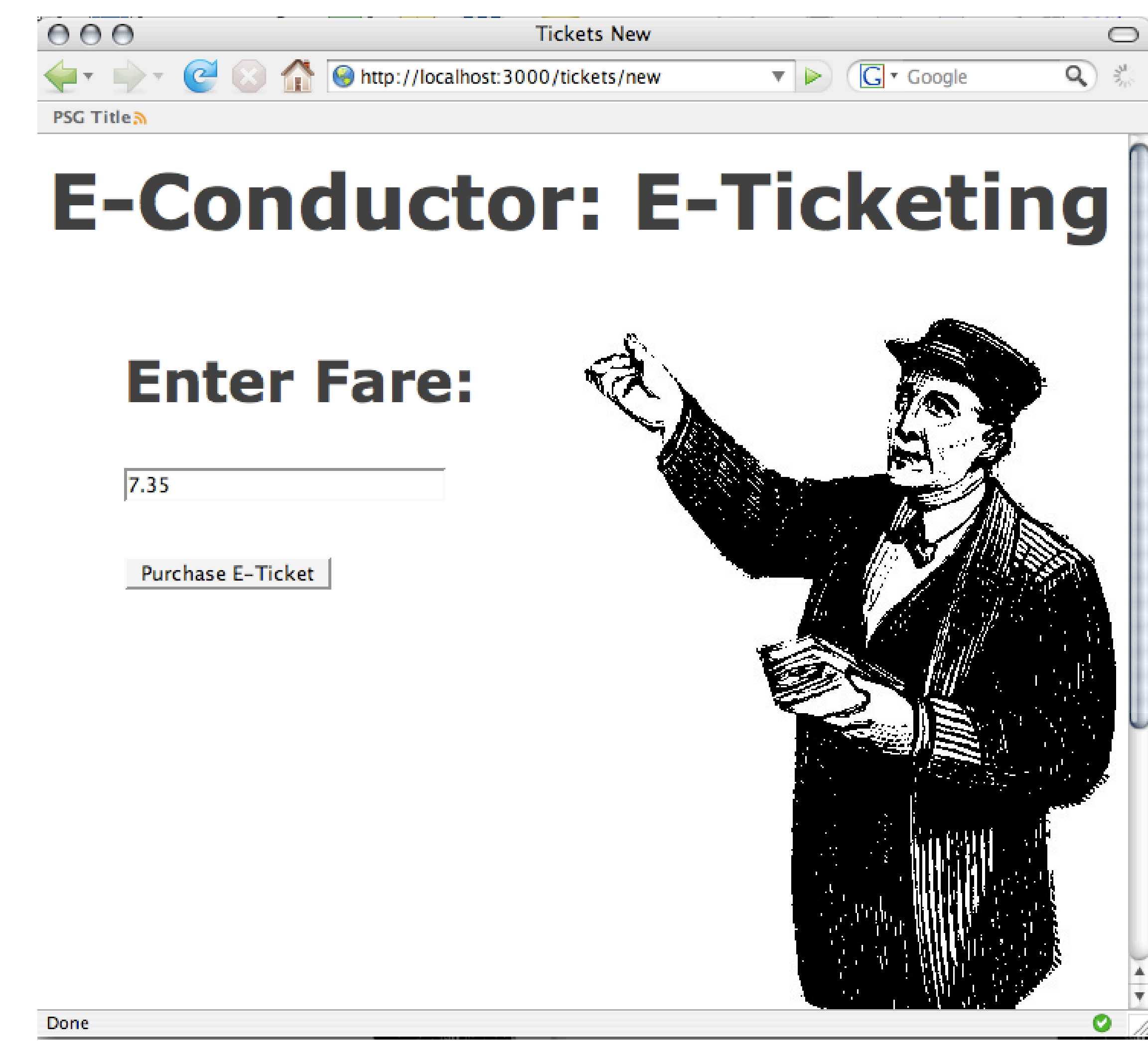
Graceful Degradation with `respond_to`

```
def create
  @ticket = Ticket.new(params)
  ticket.save
  respond_to do |format|
    format.html {redirect_to ticket_url(@ticket)}
    format.js
  end
end
```

JavaScript



HTML



Conductor drawing from http://www.rrhistorical.com

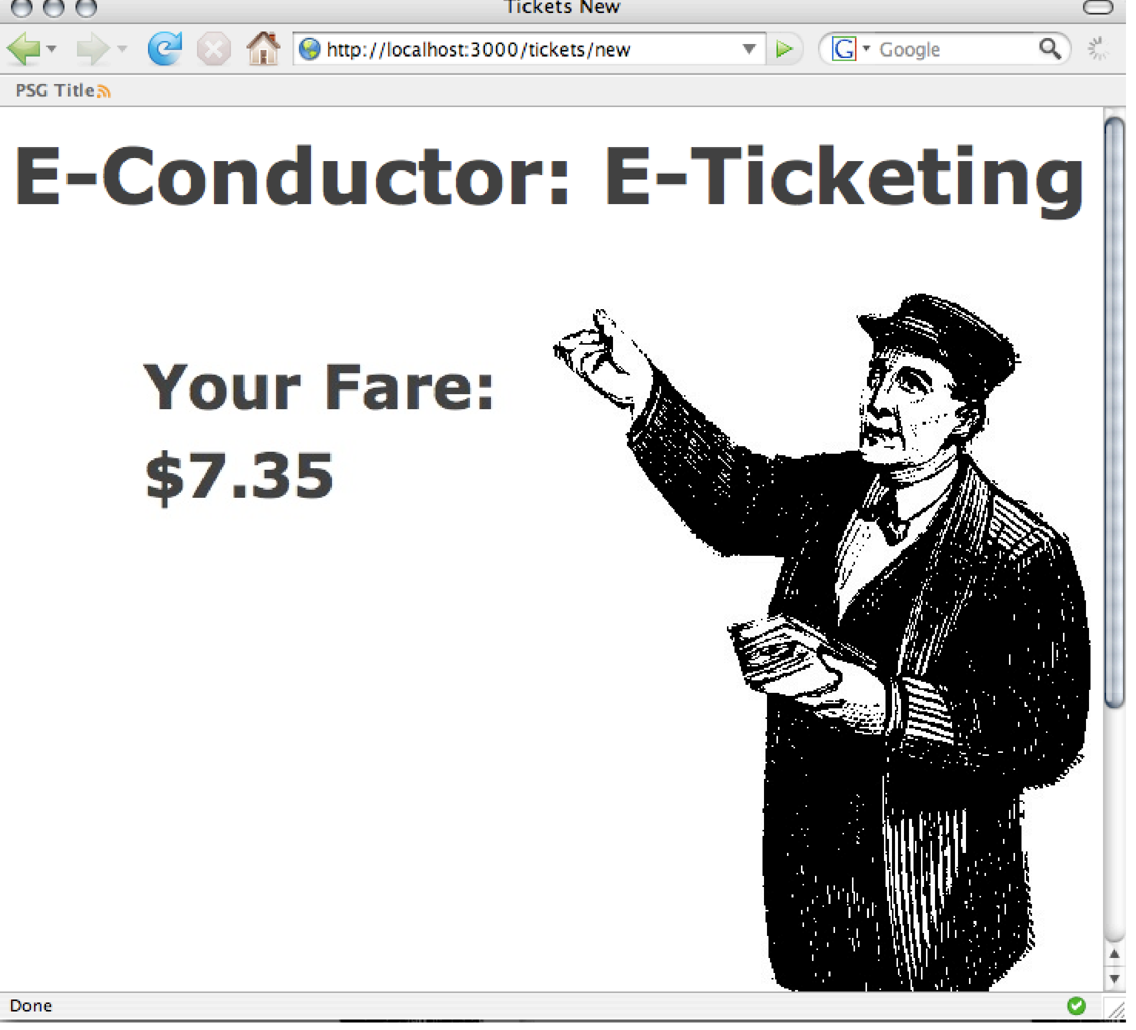
6

temporarily highlights the text in yellow ...

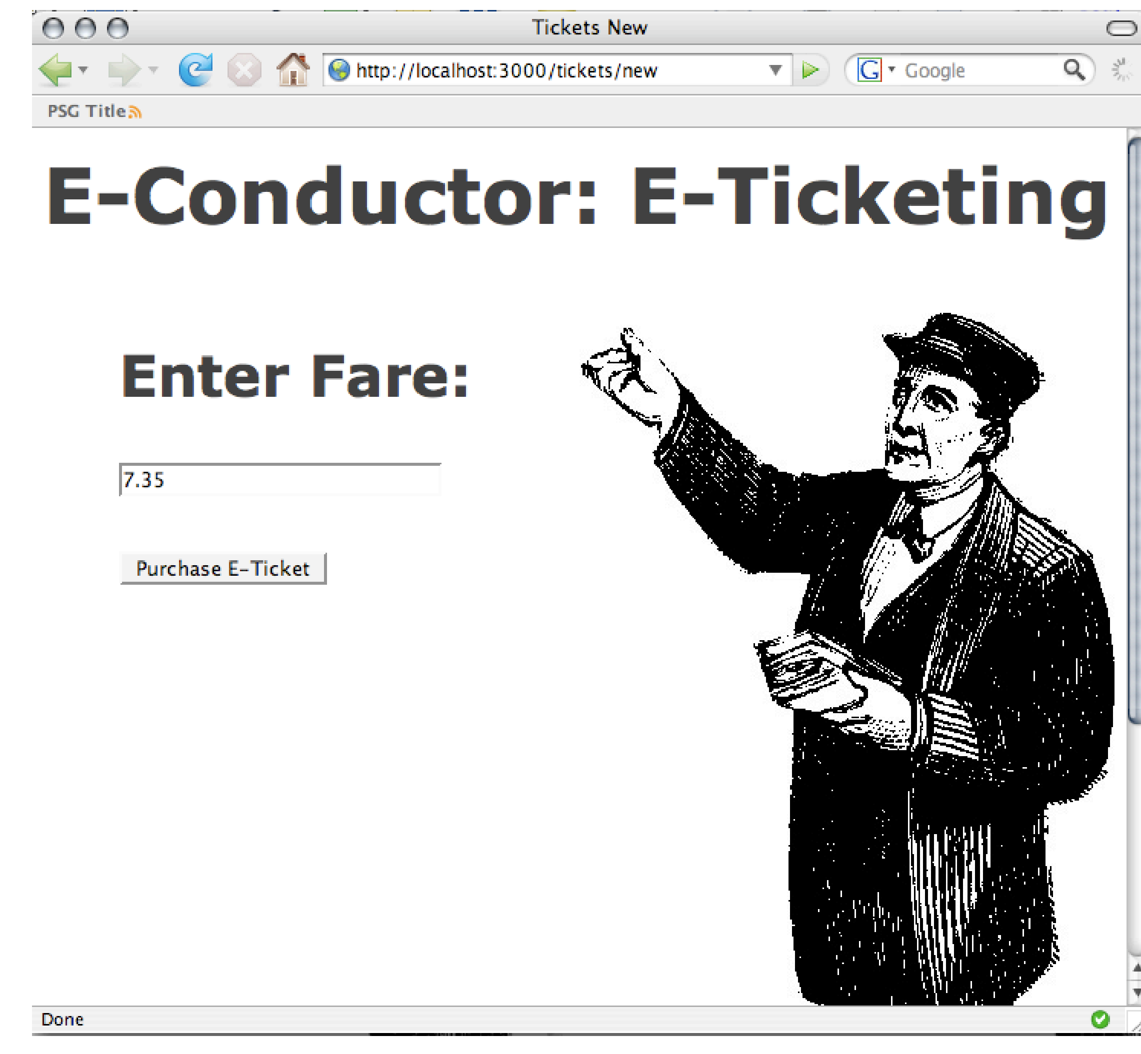
Graceful Degradation with `respond_to`

```
def create
  @ticket = Ticket.new(params)
  ticket.save
  respond_to do |format|
    format.html {redirect_to ticket_url(@ticket)}
    format.js
  end
end
```

JavaScript



HTML



Conductor drawing from http://www.rrhistorical.com

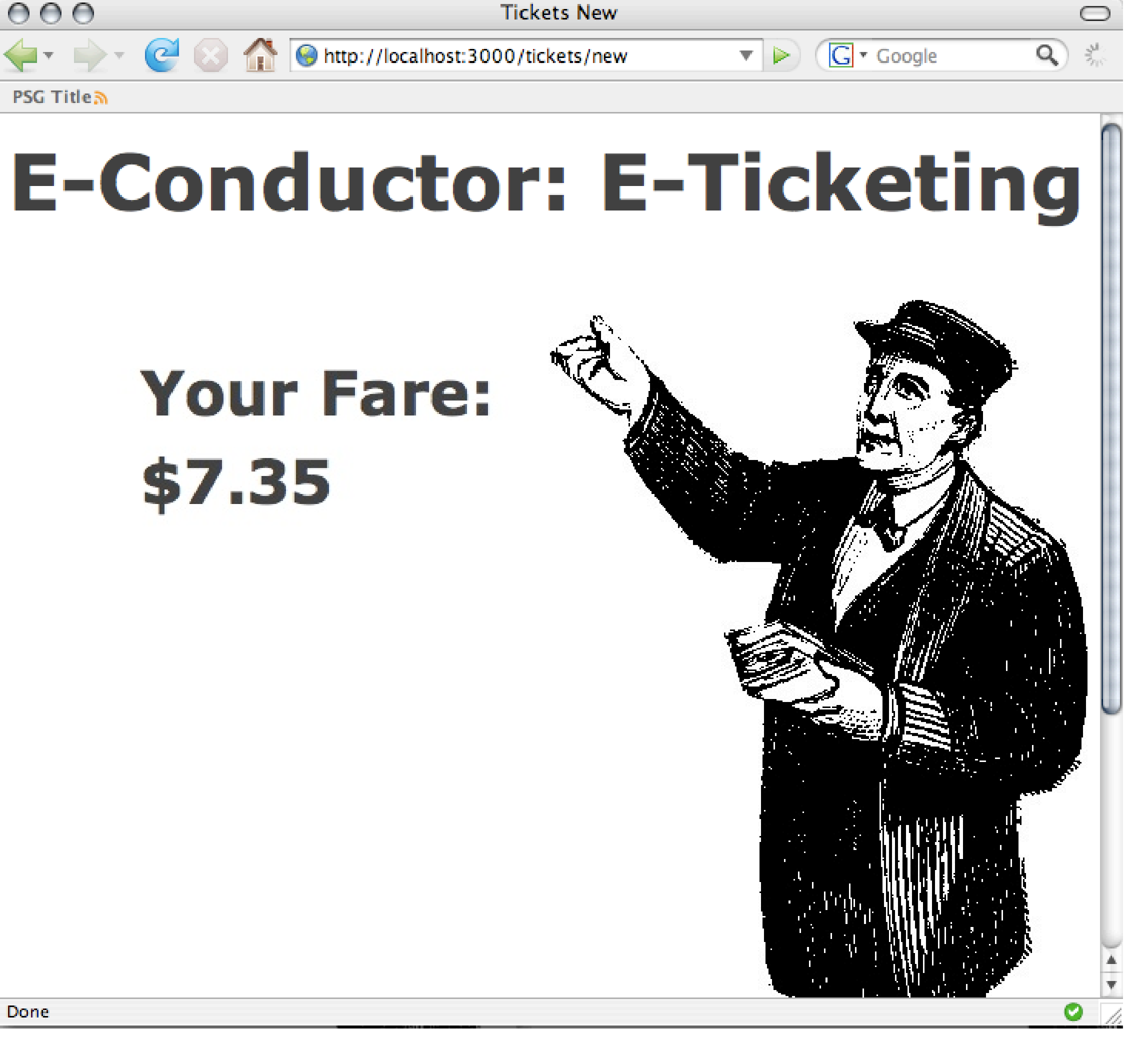
...to give the user clear feedback that the transaction was processed.

If the client is JavaScript-challenged, the user will be redirected to a page that ...

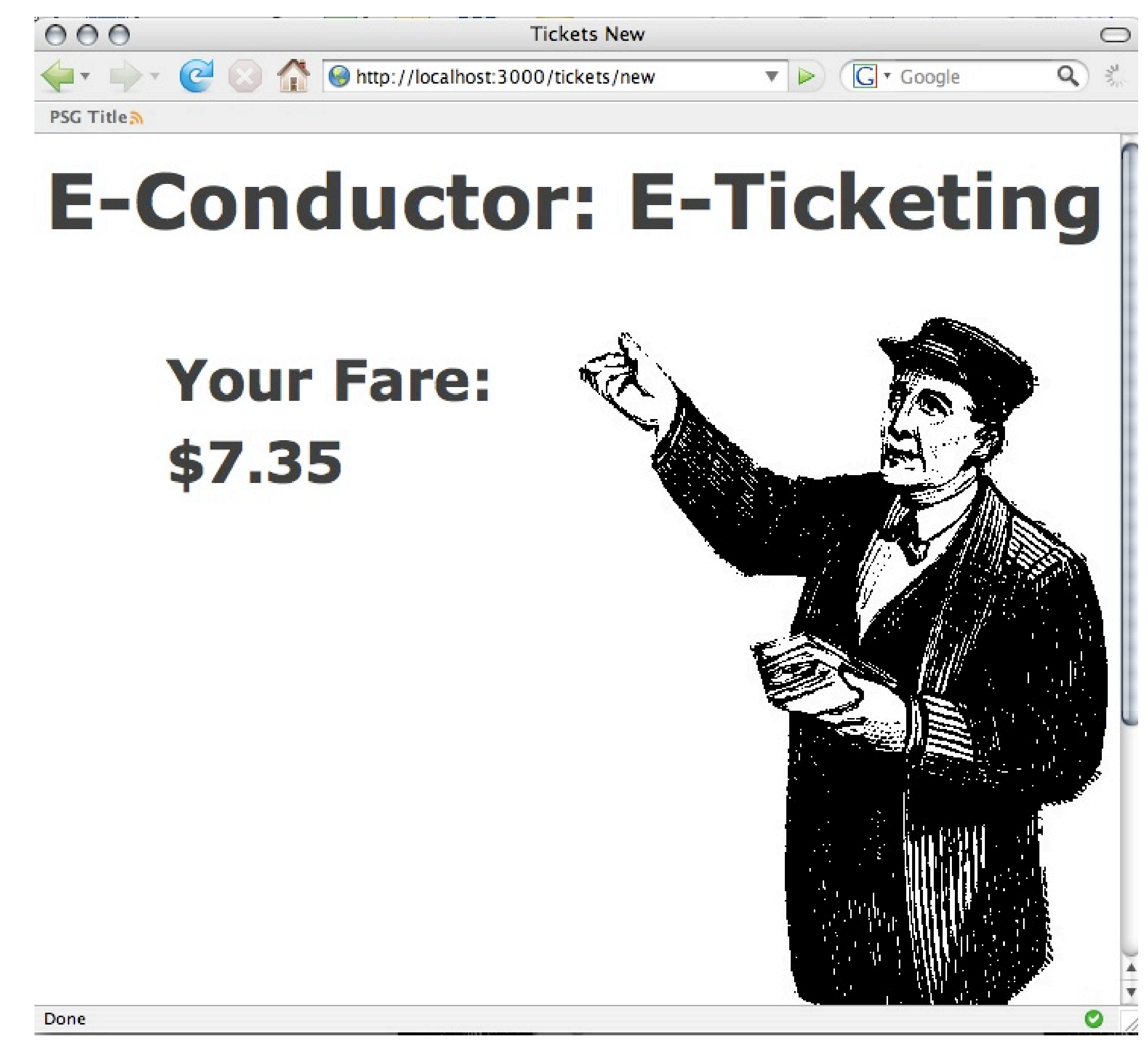
Graceful Degradation with respond_to

```
def create
  @ticket = Ticket.new(params)
  ticket.save
  respond_to do |format|
    format.html {redirect_to ticket_url(@ticket)}
    format.js
  end
end
```

JavaScript



HTML



Conductor drawing from http://www.rrhistorical.com

8

...displays the ticket value as plain text.

Graceful Degradation with `respond_to`

```ruby
def create
  @ticket = Ticket.new(params)
  ticket.save
  respond_to do |format|
    format.html {redirect_to ticket_url(@ticket)}
    format.js
  end
end
```

```ruby
#create.rjs
page["enter_fare"].replace_html render :partial => "ticket"
page["enter_fare"].visual_effect :highlight
```

Your Fare:
$7.35

Your Fare:
$7.35

```html
#show.rhtml
<h1>E-Conductor: E-Ticketing</h1>
<br/><br/>
<img class="line_art" src="/images/conductor.jpg"/>
<%= render :partial=>'ticket'%>
```

Done

Done

Conductor drawing from http://www.rrhistorical.com

Here is the code in the rjs template that replaces the input field and highlights the fare, as well as the code for the page invoked by the redirect. They both use the same partial to display the fare. So between that partial and the respond_to block, a nice chunk of View and Controller code can be used regardless of whether JavaScript is enabled or supported.

Error Handling with `respond_to`

```
def create
  @ticket = Ticket.new(params[:ticket])
  respond_to do |format|
    if @ticket.save
      flash[:notice] = 'Ticket was successfully created.'
      format.html { redirect_to account_url(@ticket) }
      format.xml  { head :created, :location => ticket_url(@ticket) }
    else
      format.html { render :action => "new" }
      format.xml  { render :xml => @ticket.errors.to_xml }
    end
  end
end
```


10

I omitted error handling from the rest of my slides to conserve time and space, but this slide shows how respond_to can be used to ensure that error messages are delivered in an appropriate format. If there's an error when you try to save your ticket, the form for entering a new ticket is redisplayed (if the client had requested HTML) -- while the errors will be returned to the caller in XML format if the client is expecting an XML response.

Agenda

1. What does **`respond_to`** do?
2. Determining what format the client "wants"
3. Proper care and feeding of **`respond_to`**
   - Default formats and extensions
     - HTML, JavaScript, XML...
   - Custom formats and extensions
     - images, voice, Easter eggs
4. Preview of Ruby features, operators and idioms that are used to implement **`respond_to`**
5. Walk through the **`respond_to`** source

I just talked about what respond_to does. In short order I'll go over how Rails determines what the client "wants" and how to use respond_to in conjunction with the standard View templates that are packaged with Rails.

I will also cover custom formats, like jpegs and pngs. There was a presentation at RailsConf about using respond_to to support a custom voice MIME type. When I talk about custom extensions, I'm going to talk about how you can use respond_to to embed an Easter egg in an Rails app.

Easter Eggs?

12

You know, Easter eggs -- undocumented features in software that are usually whimsical.

Easter Eggs?

```
aok@debian:~$ apt-get moo
```


13

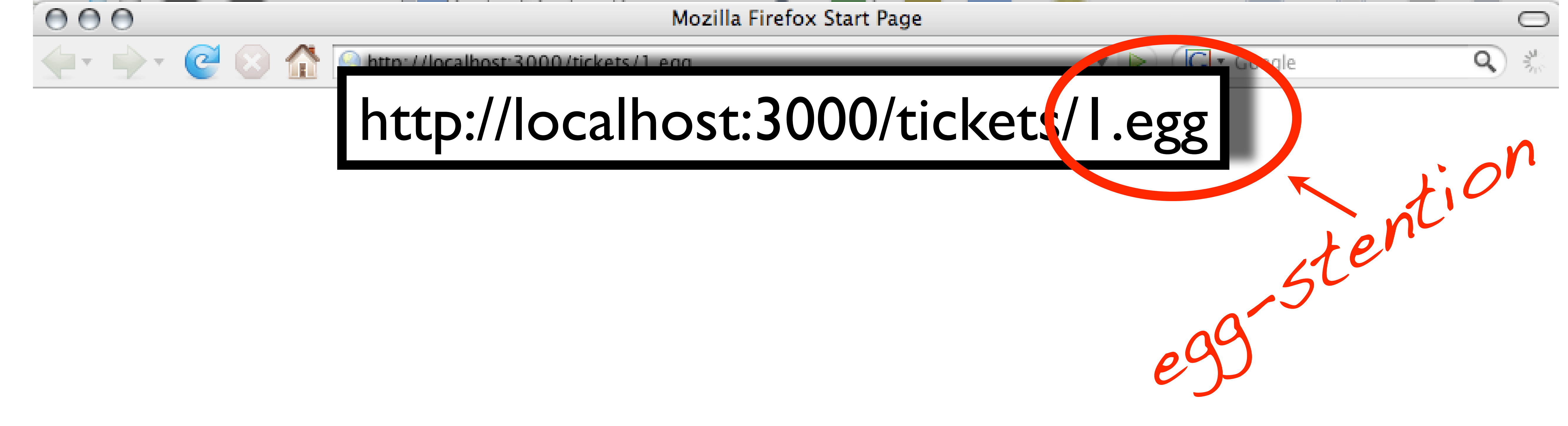
One of my favorite Easter eggs is implemented in a several Linux distributions. When you type “apt-get moo” at the command line...

Easter Eggs?

```
aok@debian:~$ apt-get moo

                 (__)
         /-------(oo)
        / |    || \/
       *   /\---/\
           ~~   ~~
....Have you mooed today?...
```

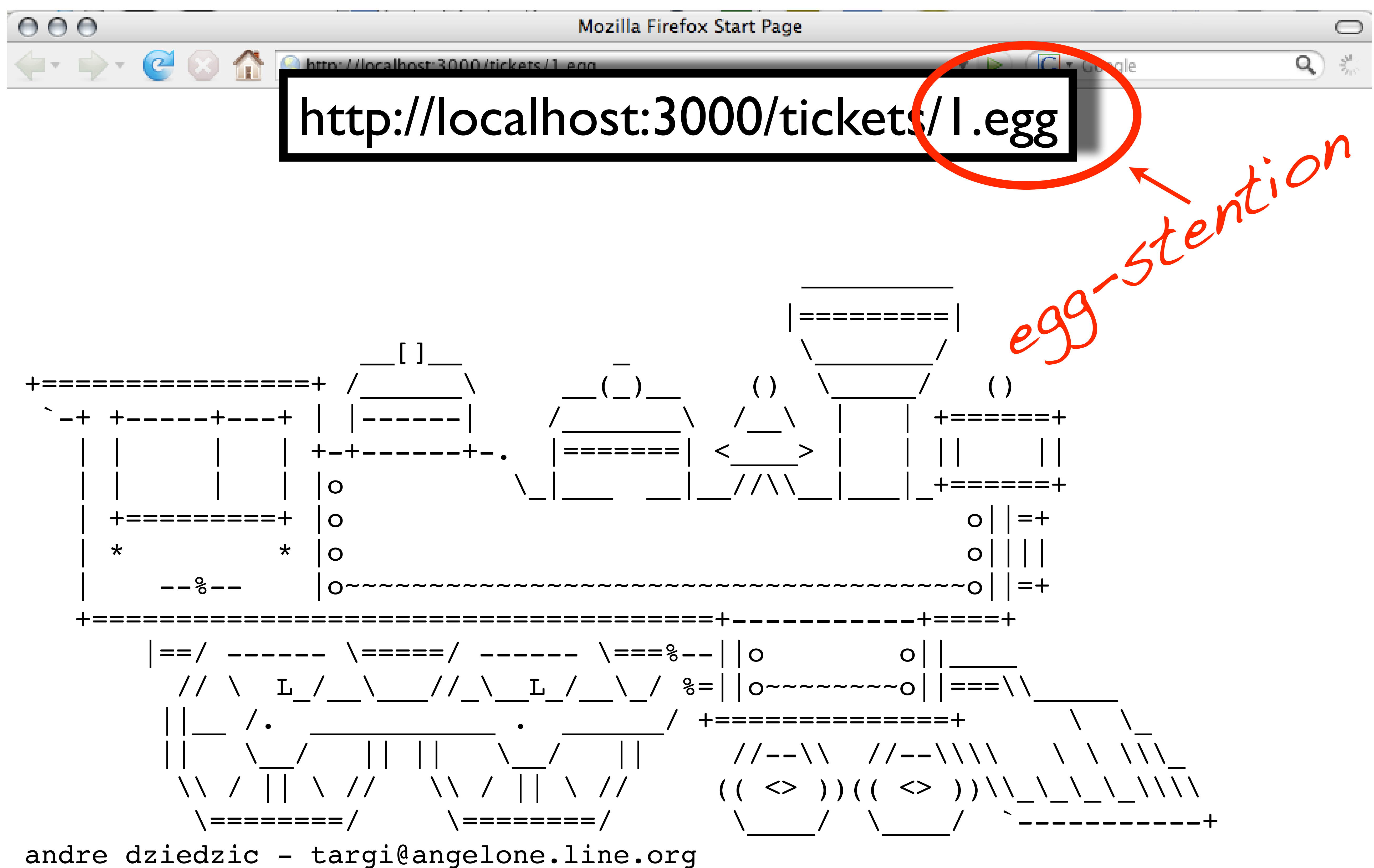

14

...you get to see this nice ASCII art cow and this phrase.



15

I'm going to demonstrate how you can modify the show action we looked at when I began this presentation -- so that when the user enters an "egg" -- "egg-stention" ...



...this ASCII art train appears.

I did not create this train, but the URL where I found it is on the lower right corner of this slide.

Agenda

1. What does **`respond_to`** do?
2. Determining what format the client "wants"
3. Proper care and feeding of **`respond_to`**
   - Default formats and extensions
     - HTML, Javascript, XML...
   - Custom formats and extensions
     - images, voice, Easter eggs
4. Preview of Ruby features, operators and idioms that are used to implement **`respond_to`**
5. Walk through the **`respond_to`** source

17

After custom formats, I will go over some of the Ruby syntax and operators that appear in the respond_to source in order to make the actual walk-through go more smoothly.

Then we will walk through all of the respond_to source.

Determining What the Client "Wants"

Request URLs with Format Tacked On

- `http://localhost:3000/train_stations/1.xml`
  - `params[:format]== "xml"`
- `formatted_train_station_path(1, :xml)`
  - `generates /train_stations/1.xml`

18

How does the app know what the client wants?

Rails provides support for specifying the preferred response format by tacking it onto the request URL. This slide shows the URL mapped to show for a TrainStationsController -- with an .xml extension tacked on. When Rails processes a request like this, a key\value pair is deposited in the params hash accessible to the Controller with format as the key and the extension, in this case, "xml", as the value.

There is a family of dynamic methods, prefixed with "formatted", that generate paths that specify format in this way.

Determining What the Client “Wants”

- **Request URLs with Format Tacked On**
  - `http://localhost:3000/train_stations/1.xml`
    - `params[:format]== ”xml”`
  - `formatted_train_station_path(1, :xml)`
    - `generates /train_stations/1.xml`
- **Format Can Be Passed as a Parameter**
  - `http://localhost:3000/train_stations/1?format=xml`

19

Format can also be passed as a parameter in the traditional manner.

Determining What the Client "Wants"

Request URLs with Format Tacked On

- `http://localhost:3000/train_stations/1.xml`
  - `params[:format]== "xml"`
- `formatted_train_station_path(1, :xml)`
  - `generates /train_stations/1.xml`

Format Can Be Passed as a Parameter

- `http://localhost:3000/train_stations/1?format=xml`

Inspecting the Request Accept Header

- `text/javascript, text/html, application/xml, text/xml, */*`

20

If there's no format parameter in the params Hash, Rails then inspects the Accept header. If more than one type is specified in the Accept header, Rails determines which one should be given priority using rules based on the HTTP spec. The Accept header shown here is used by the Prototype JavaScript library when it issues an Ajax request. If no Accept header is provided, or if the header is "*/*" (which is what Safari provides), Rails goes with whatever format is referenced first in the respond_to block.

Specifying Which Formats the Server Supports

```
def show
  @train_station = TrainStation.find(params[:id])
  respond_to do |format|
    format.html
    format.xml { render :xml => @train_station.to_xml }
    format.rss  { render :action => "rss.rxml" }
    format.atom { render :action => "atom.rxml" }
  end
end
```

respond_to	If the client "wants"...	In the respond_to block ...	Respond by...
Default formats: Default behavior	HTML XML JavaScript	format.html format.xml format.js	render [action].rhtml render [action].rxml render [action].rjs
Default formats: Custom behavior	HTML	format.html {block}	call {block}
Custom formats: Custom behavior	RSS	format.rss {block}	call {block}

21

Its pretty easy to tell how to use respond_to just by looking at examples, but I want to go over the rules briefly. This slide shows how to use respond_to to specify which response formats a Controller action supports and also what code should be executed when the client requests a particular format.

In the first row, I refer to to HTML, XML and JavaScript as default formats because Rails provides default handling for each of them. Rails is packaged with corresponding view templates for each of these formats (rhtml for html, rxml for xml and rjs for JavaScript), and the default handling for each of these formats is to render a template with the corresponding template extension and a base filename that matches the name of the action. So, the statement "format.html" in the sample respond_to block on this slide indicates that if the client requests HTML, the show.rhtml template in the train_stations directory should be rendered. To override the default logic with custom logic for one of the default formats, supply a block that contains that custom logic. Based on the respond to block in this example, the application would not render an rxml template if the client requests XML -- but instead would return the output from calling to_xml on the TrainStation instance. The syntax for custom formats mirrors the syntax for default formats linked to custom actions. You need a statement in the respond_to block to represent the custom format and you need to provide a block with the custom logic.

Custom Extensions & Formats: Easter Eggs

```
def show
  @train_station = TrainStation.find(:all)
  respond_to do |format|
    format.html
    format.xml { render :xml => @train station.to_xml }
    format.egg { render_and_stream_train }
  end
end

def render_and_stream_train
  ascii_train = "                                          __________         \n" +
    "                                         |==========|       \n" +
    "                    __[]__        _       \\________/       \n" +
    "+================+/______\\     __(_)__     ()   \\______/    () \n" +
    "`-+ +-----+---+ | |------|    /________\\  /__\\  |     |  +======+ \n" +
    "  | |     |   | +-+-------+-.  |=======| <____> |     | ||     || \n" +
    "  | |     |   | |o         \\_|___  __|__//\\\\__|____|_+======+ \n" +
    "  | +=========+ |o                                  o||=+  \n" +
    "  | *         * |o                                  o|||| \n" +
    "  |    --%--    |o~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~o||=+ \n" +
    "  +==================================+-----------+=====+ \n" +
    "      |==/ ------ \\=====/ ------ \\===%--||o        o||____ \n" +
    "        // \\  L_/__\\___//_\\__L_/__\\_/ %=||o~~~~~~~~o||===\\______ \n" +
    "       ||__ /.  ___________ .  ______/ +==============+    \\    \\_ \n" +
    "       ||    \\__/   || ||   \\__/    ||     //--\\\\  //--\\\\\\\\ \\ \\  \\\\\\_ \n" +
    "        \\\\ / || \\ //   \\\\ / || \\ //     (( <> ))(( <> ))\\\\_\\_\\_\\_\\\\\\\\ \n" +
    "         \\========/     \\========/       \\____/  \\____/  `-----------+      \n" +
    " andre dziedzic - targi@angelone.line.org "
  send_data ascii_train,:type=>'text\plain',:disposition=>'inline'
end
```

```
Mime::Type.register "easter_egg", :egg, %w(undocumented/whimsy)
```


22

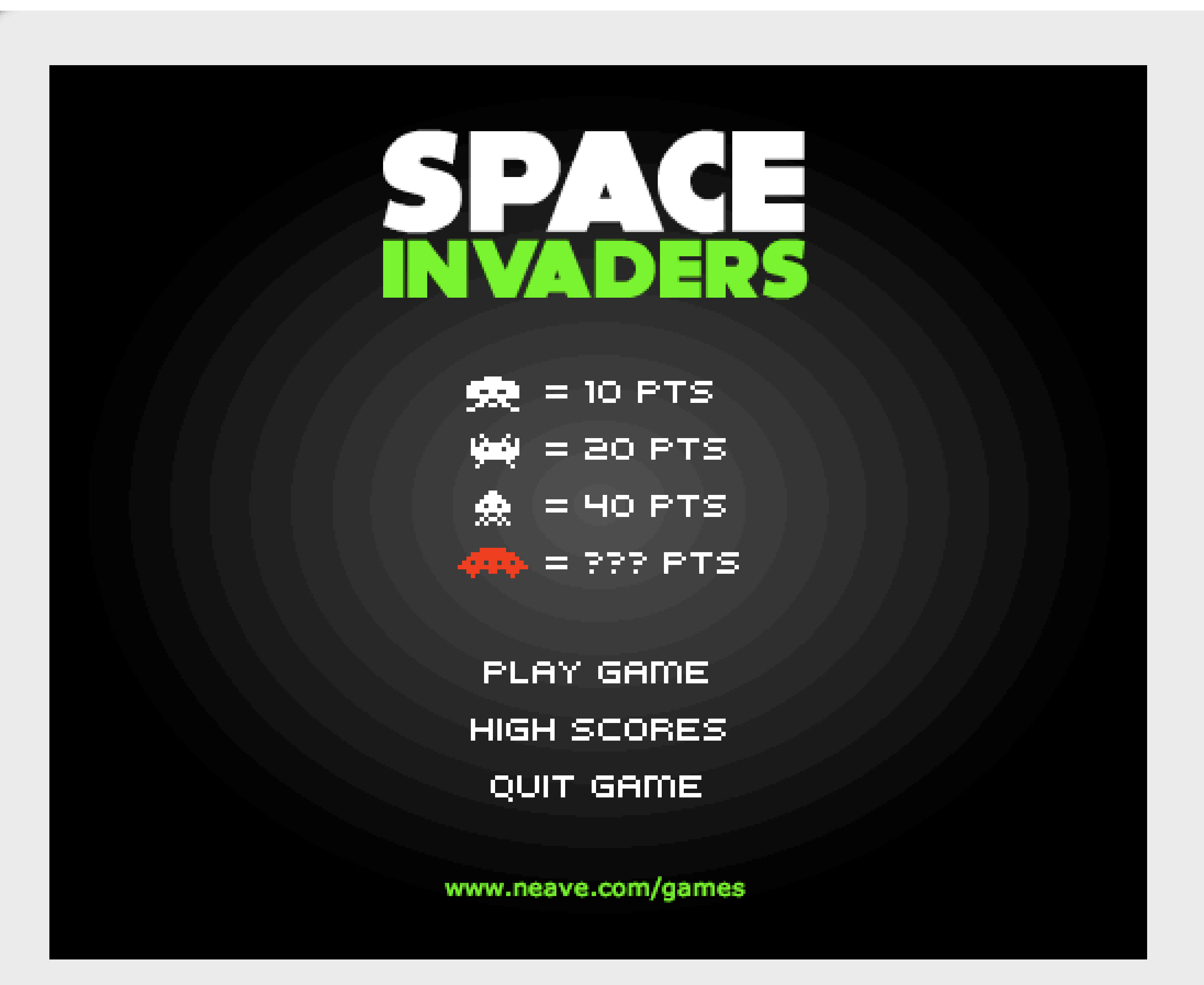
Using some custom formats requires the additional step of registering the custom type in environment.rb. Although Rails only provides default handling for 3 formats (HTML, XML and JavaScript), there are pre-defined Mime::Type constants in the Rails core source for numerous other formats -- including ical, rss, atom, and text. Image types need to be registered. Because, not surprisingly, there is no pre-defined MIME::Type constant for Easter eggs, I needed to add this one-liner to environmet.rb in order to implement my Easter egg. We'll look at the source code for Mime::Type.register later. For now, you can see that I needed to supply it with a name, a symbol for the extension, and any synonyms I want Rails to know about.

Here is the show action we looked at when I started the presentation -- modified to support an Easter egg. The line highlighted in red indicates that if the client is interested in the Easter egg, the render_and_stream_train method should be invoked. As you can see, that deposits the ASCII characters that constitute the train into a String variable and passes that to send_data to stream it out to the client.

To support an image-based representation of a TrainStation, I would need to register my image type, png, for example. I would need to add “format.png” to the respond_to block, along with a block that takes care of rendering the image, perhaps leveraging send_data.

The Spaceship Operator: <=>

```
class SpaceInvader

  attr_accessor(:name,:value)

  def initialize(name,value)
    @name = name
    @value=value
  end

  def <=>(other)
    value <=> other.value
  end

  def to_s
    name
  end

end
```



Space Invaders graphic: www.neave.com

23

Now we're getting into the part of the presentation that previews some of the Ruby operators we will encounter in the respond_to source.

The <=> operator, commonly referred to as the spaceship operator, is used to customize sort order. By default when you call sort on an Array of numbers, the numbers are ordered in ascending order. But let's say you wanted the numbers to represent playing cards. You might want "1" to represent an ace. You could override the spaceship operator for your card class to specify that 1 should be considered greater than all the other numbers when sorting.

This code shows the definition of a simple SpaceInvader class for a video game. Each SpaceInvader has a name and a point value (how many points you get when you hit it with a piping symbol you launched). Here the spaceship operator is overridden to specify that sort order should be based on point value ascending.

The Spaceship Operator: <=>

```
class SpaceInvader

  attr_accessor(:name,:value)

  def initialize(name,value)
    @name = name
    @value=value
  end

  def <=>(other)
    other.name <=> name
  end

  def to_s
    name
  end

end
```

Space Invaders graphic: www.neave.com

24

Here is the same class with the spaceship operator overridden to base sort order on name in reverse alphabetical order.

The "Threequal" Operator: ===

```
class Person

  def initialize(name)
    @name = name
  end

  def ===(other)
    if (other.name == name)
      return true
    end
    super
  end

  attr_accessor :name

end
```

```
class Cat

  def initialize(name)
    @name = name
  end

  def ===(other)
    if (other.name == name)
      return true
    end
    super
  end

  attr_accessor :name

end
```


25

The === operator is commonly referred to as the threequal operator. It's the operator that Ruby consults when evaluating the "when" clause of a case statement. By default, it returns the same value as the == operator.

Since we'll be looking at Ruby and Rails features that are sometimes referred to as magical , I thought it would be fitting to look at some code examples based on the Harry Potter series. This is the first of these.

Professor McGonagall teaches transfiguration at the Hogwarts Academy of Witchcraft and Wizardry. Transfiguration is the art of changing something into something else. One of the most difficult kinds of transfiguration to master is turning a person into an animal -- and Professor McGonagall has been know to turn herself into a cat.

Normally one would not say that a person == (is equal to) to a cat, but here threequal operator is overridden for the Person class (which support a name property) and the Cat class (which also supports a name property) -- so that a Person and a Cat are considered threequal if they answer to the same name.

The “Threequal” Operator: ===

```
PROFESSOR_MCGONAGALL = Person.new(“Professor McGonagall”)
```

```
cat_on_privet_drive = Cat.new(“Professor McGonagall”)
```

```
cat_on_privet_drive === PROFESSOR_MCGONAGALL
```

```
=> true
```

```
case cat_on_privet_drive
  when PROFESSOR_MCGONAGALL
     puts "The cat is Professor McGonagall"
  default
    puts "The cat is a garden-variety cat."
end
```

```
=> The cat is Professor McGonagall.
```


26

The constant in the first line here represents Professor McGonagall. The next line represents her turning herself into a cat. The variable is called "cat_on_privet_drive" because when Professor McGonagall is first introduced in the first chapter of the first Harry Potter book, she is in cat form outside the house on Privet Drive where Harry's uncle, aunt and cousin live.

When Professor Dumbledore, the headermaster at Hogwarts, famously addresses the cat as "Professor McGonagall", he is basically asserting that the cat and Professor McGonagall are threequal.

In the alternative, he could have used a case statement like the one shown here. Since the case statement defers to the === operator , the first “when” clause would evaluate to true.

Being Able to Grok Blocks & Procs

Some Blocks

```
{ puts “All aboard!” }
```

```
{ puts “All aboard!”; puts “Watch your step!” }
```

Using the “Just Do It” Syntax

```
do
  puts “All aboard!”
  puts “Watch your step!”
end
```


27

It’s important to have some familiarity with blocks and Procs to understand how respond_to works.

A block is comprised of one or more statements, which can be grouped using either curly braces or the “do” and “end” keywords. Maybe it’s just me, but for some reason, I had a block against the “Just Do It” (do\end) syntax when I first started learning Ruby. Maybe it’s because it’s structurally so similar to do\while loops in some other languages (there is no do\while loop in Ruby).

The middle box and the bottom box both contain the same statements, using different syntax.

Being Able to Grok Blocks & Procs

Blocks Can’t Run on Their Own

```
{ puts “All aboard!”; puts “Watch your step!” }
```

```
=> SyntaxError: compile error
(irb):1: syntax error, unexpected tSTRING_BEG, expecting kDO
or '{' or '('
{ puts "All aboard!"; puts "Watch your step!" }
        ^
(irb):1: syntax error, unexpected '}', expecting $end
        from (irb):1
irb(main):002:0>
```


28

A block is not a valid statement on its own. If you enter a block at the prompt during an interactive Ruby session, you will see an error like the one shown here in the box with a dotted outline.

Being Able to Grok Blocks & Procs

Yielding to a Block

```
def method_that_yields
  yield
end

method_that_yields { puts “All aboard!”; puts “Step right up!” }

method_that_yields do
  puts “All aboard”
  puts “Step right up!”
end
```

```
=> All aboard!
   Step right up!
   All aboard!
   Stop right up!
```


29

So how can you invoke the logic in a block?

One way is to pass the block to a method that yields. You can actually pass a block to any method -- regardless of whether a block is specified in its signature. If the block contains a yield statement -- the block you passed in will be triggered by the yield.

This slide shows what it looks like when a block is passed to method_that_yields using the curly braces syntax and also using the do\end syntax. The output from the two method calls is shown in the box with a dashed border.

Being Able to Grok Blocks & Procs

Passing Argument(s) to a Block

```
def my_method(stop)
  puts stop.upcase
end
```


30

Before we look at the syntax for passing an argument to a block, I wanted to take a look a the familiar syntax for passing an argument to a method. This method outputs the name of station stop passed to it in capital letters. You can count on a conductor to shout out the name of a station as the train approaches it, and capitalizing is the lexigraphical equivalent of shouting.

Arguments are enclosed in parentheses in a method signature.

Being Able to Grok Blocks & Procs

Passing Argument(s) to a Block

```
def my_method(stop)
  puts stop.upcase
end
```

```
do |stop|
  puts stop.upcase
end
```

```
{ |stop| puts stop.upcase }
```


31

Here are a couple of blocks that perform the same function as my_method. You specify the name of a block argument by placing it between 2 piping symbols, instead of using a pair of parentheses. You use the piping symbol goal posts regardless of whether you use the “do\end” delimiters or the curly braces.

Being Able to Grok Blocks & Procs

Passing Argument(s) to a Block

```
def my_method(stop)
  puts stop.upcase
end
```

```
do |stop|
  puts stop.upcase
end
```

```
{ |stop| puts stop.upcase }
```

```
def red_line_train_ride
  stations = [“Woodley Park”, “National Zoo”, “Dupont Circle”]
  for station in stations
    yield(station)  <--------------------
  end
end

red_line_train_ride do |stop|
  puts stop.upcase
end
```

```
=> WOODLEY PARK
   NATIONAL ZOO
   DUPONT CIRCLE
```


32

When a block takes one or more arguments, you pass them to yield, which in turn passes them to the block.

Here we pass our block that shouts out train stops to red_line_train_ride, a brief programmatic ride on the Washington, DC’s Metroliner. The logic loops through an Array of station stop names, passing each to yield, which passes them on them to the block.

The box with the dashed border shows the output.

Being Able to Grok Blocks & Procs

Binding Blocks to Variables

A Block Can’t Be Bound to a Variable “As Is”

```
yell = { |station_name| puts station_name.upcase }
```


33

An unadorned block can’t be assigned to a variable.

Being Able to Grok Blocks & Procs

Binding Blocks to Variables

```
yell = lambda { |stop| puts stop.upcase }
shout = Proc.new { |stop| puts stop.upcase }

yell.call("Dupont Circle")

shout.call "Woodley Park"
```

```
=> DUPONT CIRCLE
   WOODLEY PARK
```


34

If you want a named reference for the logic that constitutes a block -- the block needs to be turned into a Proc. Passing the block to Proc.new or lambda will turn it into a Proc. Calling "call" on a Proc invokes the logic in its underlying block.

There are some differences between using Proc.new and lambda. If Proc.new was used, a return statement encountered in a block will boot the logic flow out of the enclosing method -- while if lambda had been used program execution would return to the code following the block invocation. You also get stricter arity checking with lambda. With Proc.new Ruby will not ensure that the number of arguments passed to call matches the number of arguments in the block definition.

Just as you can't assign a block to a variable directly, you can't add a block to an Array or a Hash. You can, however, store Procs in Arrays and Hashes.

Being Able to Grok Blocks & Procs

Calling "`call`" on a Block You Call Something

```
def method_that_calls(&block_to_call)
  block_to_call.call
end

method_that_calls { puts "All aboard!"; puts "Step right up!" }

method_that_calls do
  puts "All aboard"
  puts "Step right up!"
end
```

```
=> All aboard!
   Step right up!
   All aboard!
   Step right up!
```


35

You can also refer to a block by name if you include it in the argument list for a method. It should always be the last argument in the argument list, and it needs to be prefixed with an ampersand.

When a block is passed to a method in this manner, it becomes a Proc within the method, and you can call "call" on it to invoke its logic.

Being Able to Grok Blocks & Procs

What's in a Name?

```
respond_to do |format|
  format.html
  format.xml {render :xml => @train_station.to_xml}
end
```

```
respond_to do |wants|
  wants.html
  wants.xml {render :xml => @train_station.to_xml}
end
```

```
respond_to do |accepts|
  accepts.html
  accepts.xml {render :xml => @train_station.to_xml}
end
```

```
respond_to do |type|
  type.html
  type.xml {render :xml => @train_station.to_xml}
end
```


36

One of the reasons I included a number of slides showing how blocks are structured is to help orient those who are new to Ruby. Another reason is that in preparation for looking at the respond_to source, I wanted everyone to start thinking of respond_to in terms of passing blocks to methods and passing arguments to blocks. I don't think most developers, even experienced developers, think of respond_to in those terms. I think that most developers think of respond_to as domain specific language for handling response formats. You don't look at a respond_to block and think "This respond_to block takes one argument, called format. A method named 'html' is called on format. Then 'xml' is called on format, and a block is passed to the 'xml' method".

In actual fact the respond_to block is not so different from the blocks we have been looking at. The block argument does not have to be called 'format'. It can be called anything and still be syntactically correct. I've actually seen several different names for this argument. The RDoc for Rails and the scaffold_resource generator uses 'format'. The Pragmatic Rails book uses 'accepts'. In one of his early blog entries about respond_to, DHH used 'type'. I have seen 'wants' on a few blogs. Which of these do you think works best for a return format DSL?

Being Able to Grok Blocks & Procs

Blocks Are Closures

```
hermione=387072000

def the_recent_past(&time_turner)
  hermione=387064800
  puts "Hermione in the recent past: #{hermione}."
  traveller_hermione = eval("hermione", time_machine.binding)
  puts "Time travelling Hermione: #{traveller_hermione}."
end

the_recent_past {puts “This is a time turner.”}
```


37

Closures are language constructs that link logic with the environment or the application state. Because a block is a closure, it is linked to the bindings for all the variables that were in scope at the time the block was defined. There are Ruby functions you can use to interrogate a block to find out the value of any variable that existed at the time the block was declared, regardless of whether it is referred in any of the statements that constitute the block.

I'll use another Harry Potter example to show how closures can be used. There's a student who goes to school with Harry named Hermione Granger, who is a top student and wants to take all of the classes that are offered -- even those that are scheduled at the same time. So Headmaster Dumbledore obliges by giving her a device called a time turner that allows her to go back in time. Let's say transfiguraton and potions are both scheduled from 10 AM unto noon on Mondays. She can go to transfiguration class. When it is over at noon, she can use the time turner to go back a couple of hours. She can then go to potions class.

This code expresses her time travel routine programatically. The first line defines 'hermione'. She a little more than 12 years old -- in seconds. The method the_recent_past, which takes a block called 'time turner' , serves to take Hermione back in time. If you do the math, you can see that the local 'hermione' in the_recent_past is two hours younger than the 'hermione' at the present time, defined on the top line. The eval call is where the magic happens. It's what enables us to access the time-traveling 'hermione' from the present in the_recent_past.

Being Able to Grok Blocks & Procs

Blocks Are Closures

```
hermione=387072000

def the_recent_past(&time_turner)
  hermione=387064800
  puts "Hermione in the recent past: #{hermione}."

   traveller_hermione =
     eval("hermione", time_turner.binding)

  puts "Time travelling Hermione: #{traveller_hermione}."
end

the_recent_past {puts “This is a time turner.”}
```


38

Let’s take a closer look at eval. You pass eval a String -- and eval treats that String like source code -- evaluating its return value by way of executing it. It literally turns words into action. It’s very powerful. You can use Ruby’s many String manipulation methods to build a line or lines of code, and then run that code using eval.

If you pass eval the variable bindings that are linked to a block (which you can access by simply calling ‘binding’ on the block), eval will use the variable assignments linked to to that block when it needs to determine a variable value.

If you pass eval a variable name, it will return that value of that variable. And if you pass eval a variable name and the bindings associated with a particular block -- it will return the value the variable had at the time the block was defined.

In the last line, we actually send Hermione back in time, by calling the_recent_past and passing it a block. The block simply prints out a line identifying it as a time turner. You can see that we are not explicitly passing ‘hermione’ to the_recent_past. However, we can use eval to access the time-traveling ‘hermione’ of the present from within the_recent_past -- because the present-time ‘hermione’ is the one that was in scope when we created a block to pass to the_recent_past.

Being Able to Grok Blocks & Procs

Blocks Are Closures

```
hermione=387072000

def the_recent_past(&time_turner)
  hermione=387064800
  puts "Hermione in the recent past: #{hermione}."

   traveller_hermione =
     eval("hermione", time_turner.binding)

  puts "Time travelling Hermione: #{traveller_hermione}."
end

the_recent_past {puts “This is a time turner.”}
```

```
=> Hermione in the recent past: 387064800.
   Time travelling Hermione: 387072000.
```


39

When we run the code, the output from the diagnostic puts statements in the_recent_past show that there are 2 distinct Hermiones in the realm of the_recent_past -- the local ‘hermione’ and the time-traveling ‘hermione’.

Incidentally, One important time travel rule is that you have to be very careful not to run into your future self or the consequences for your future can be devastating. Time traveling is a low risk prospect for Hermione because she knows exactly where here future self was during the potions class time slot. All she has to do is avoid transfiguration class.

Being Able to Grok Blocks & Procs

Blocks Are Closures

```
def show
  respond_to do |format|
    format.html
    format.xml {:xml => @train_station.to_xml}
    format.egg { render_and_stream_train }
  end
end
```

The binding for this block can provide access to the `response`, the `request` and the `params` that its enclosing Controller has access to.

40

Here's a slightly more practical example. When we look at the respond_to source, we'll see that the variable bindings linked to the block passed to respond_to get passed to a class that needs to leverage eval in order to inspect the params Hash and the request, and that also sets attributes on the response.

41

The `respond_to` Source

```
def respond_to(*types, &block)
  raise ArgumentError, "respond_to takes either types or a block..."
    unless types.any? ^ block

  block ||= lambda {|responder|types.each {|type| responder.send(type)}}

  responder = Responder.new(block.binding)

  block.call(responder)

  responder.respond
end
```

from ActionController::MimeResponds::InstanceMethods

42

Here is the respond_to source in its entirety. In the slides that follow, the code snippets from this method will be displayed with white text on a black background.

As you can see it take either *types or a block, although you rarely see examples of passing *types to respond_to in articles, books or blogs.

The splat operator (*) in the "*types" indicates that any number of types can be passed to respond_to, and that Ruby will automatically deposit them in an Array that can be referenced as "types".

The first line of the method indicates that the method accepts either *types or a block, not both. The caret (^) indicates an "exclusive or" condition. The expression evaluates to true if one, and only one, of the two conditions is true -- not if either is true. So the ArgumentError will be raised unless only one argument is supplied.

The `respond_to` Source

```
def respond_to(*types, &block)
  raise ArgumentError, "respond_to takes either types or a block..."
    unless types.any? ^ block
  block ||= lambda {|responder|types.each {|type| responder.send(type)}}
  responder = Responder.new(block.binding)
  block.call(responder)
  responder.respond
end
```

from ActionController::MimeResponds::InstanceMethods

43

So now we're finished with the first line and can move on. The next line creates a block, based on *types, if the caller supplied *types in lieu of a block. Once that block is defined, the logic is the same regardless of whether *types or a block were supplied.

Note that regardless of whether "block" is supplied or populated with the code on the second line, it's actually a Proc -- despite being named "block". As explained earlier, Ruby transforms a block into a Proc when it is named in the method signature, and lambda returns a Proc. That being said, I'm going to refer to "block" as a block throughout the rest of the presentation. I hope that does not offend anyone's sensibilities. For all intents and purposes, "block" is a block. You can access the variable bindings it is linked to (calling "binding" on a Proc returns the variable bindings linked to its underlying block) and you can invoke its logic. I'm also going to refer to the block called "block" as "the respond_to block".

Before we move on to the meat of respond_to, let's take a quick look at an example Controller action that passes *types to respond_to.

Passing *types to respond_to:
Creating a block based on *types if one isn't supplied

```
def show
   @train_station = TrainStation.find(params[:id])
   respond_to(:html, :xml)
end
```

```
block ||=
lambda {|responder| types.each {|type| responder.send(type)}}
```

```
types == [:html,:xml]
```

```
responder.send(:html)
responder.send(:xml)
```

```
responder.html
responder.xml
```

```
def show
   @train_station = TrainStation.find(params[:id])
   respond_to do |format|
     format.html
     format.xml
   end
end
```


44

In this sample show action, :html and :xml are passed to respond_to instead of a block. By virtue of the splat operator (*), Ruby deposits these values in an Array called ‘types’. Rails populates the block variable by creating a Proc via lambda. The block passed to lambda takes an argument named “responder”. The block contains logic to loop through the types Array, calling “send” on “responder” and passing “send” the type. Calling “send” with a method name as an argument is equivalent to calling the method -- so “responder.send(:html)” is equivalent to “responder.html”.

As it turns out, the block that gets constructed when *types are supplied is functionally equivalent to passing a block that references only default types and does not specify custom behavior for those types. Passing *types is only a viable option if the action only supports default return types -- and can use the default behavior for those formats.

The variable bindings associated with the block constructed if *types is supplied include params, request, and response because Rails adds respond_to to ActionController via Ruby’s mixin mechanism.

The `respond_to` Source

```
def respond_to(*types, &block)
  raise ArgumentError, "respond_to takes either types or a block..."
    unless types.any? ^ block

  block ||= lambda {|responder|types.each {|type| responder.send(type)}}

  responder = Responder.new(block.binding)

  block.call(responder)

  responder.respond
end
```

from ActionController::MimeResponds::InstanceMethods

45

Now we've covered the first two lines and can move on to the third. Here, a Responder is created. Its constructor takes the binding for the block that specifies which formats the action supports and corresponding behavior for each action.

Creating an ActionController::MimeResponds::Responder

```
responder = Responder.new(block.binding)
```

```
class Responder

  DEFAULT_BLOCKS = [:html, :js, :xml].inject({}) do |blocks, ext|
    ...# construct default blocks for rails standard templates
  end

  def initialize(block_binding)
    ...# determines which response type is preferred
  end

  def custom(mime_type, &block)
    ...# creates an array of responses that correspond to types
  end

  def method_missing(symbol, &block)
    ...# calls custom
  end

  def respond
    ...# send the client a response in the expected format
  end

end
```


Here is a skeletal version of the Responder class. The initialize method for Responder is where Rails determines which response type the client wants.

Determining Which MIME Type Gets Priority While Initializing the Responder

```
class Responder
  ...
  def initialize(block_binding)
    @block_binding = block_binding
    @mime_type_priority =
    eval(
      "(params[:format] && Mime::EXTENSION_LOOKUP[params[:format]]) ? " +
      "[ Mime::EXTENSION_LOOKUP[params[:format]] ] : request.accepts",
      block_binding
    )
    @order     = []
    @responses = {}
  end
  ...
end
```

```
@mime_type_priority =
  eval(
       "(params[:format] && Mime::EXTENSION_LOOKUP[params[:format]]) ? " +
       "[ Mime::EXTENSION_LOOKUP[params[:format]] ] : request.accepts",
       block_binding
   )
```


The @mime_type_priority instance variable represents the formats that the client will accept. Later on we'll see that @order represents the formats the Controller action supports, the keys in the @responses Hash correspond to the formats the action supports, and the values in the @responses Hash are Procs that contain the response logic for the supported formats.

Based on the ternary expression passed to eval -- the @mime_type_priority will be assigned the Mime::EXTENSION_LOOKUP Hash value paired with the value of the format parameter -- if there is a format value in the params Hash and if there is a value paired with that format in the Mime::EXTENSION_LOOKUP Hash. Otherwise the value assigned to @mime_type_priority will be the return value of request.accepts.

Note that a Responder has access to params and request by virtue of the binding passed to its constructor.

Determining Which MIME Type Gets Priority
If params[:format] is supplied...

```
[Mime::EXTENSION_LOOKUP[params[:format]]]
```

```
module Mime
...
  ALL  = Type.new "*/*", :all
  TEXT = Type.new "text/plain", :text
  HTML = Type.new "text/html",:html, %w( application/xhtml+xml )
  XML  = Type.new "application/xml", :xml, %w( text/xml application/x-xml)
  ...
  SET = [ALL, TEXT, HTML, JS, ICS, XML, RSS, ATOM, YAML, JSON]

  LOOKUP = Hash.new { |h, k| h[k] = Type.new(k) unless k == "" }
  LOOKUP["*/*"]                     = ALL
  LOOKUP["text/html"]               = HTML
  LOOKUP["application/xhtml+xml"]   = HTML
  LOOKUP["application/xml"]         = XML
  LOOKUP["text/xml"]                = XML
  LOOKUP["application/x-xml"]       = XML
  ...
  EXTENSION_LOOKUP = Hash.new { |h, k| h[k] = Type.new(k) unless k == "" }
  EXTENSION_LOOKUP["html"]  = HTML
  EXTENSION_LOOKUP["xhtml"] = HTML
  EXTENSION_LOOKUP["txt"]   = TEXT
  EXTENSION_LOOKUP["xml"]   = XML
  ...
end
```


48

If there's a format value in the params Hash, Rails looks up the value that corresponds with that format value in the Mime::EXTENSION_LOOKUP Hash. EXTENSION_LOOKUP is defined near the bottom of this abbreviated version of the Mime module. You can see that the value for each format type is a constant named for that format type. These constants are defined near the top of the module. Each one is a Mime::Type.

Determining Which MIME Type Gets Priority If params[:format] is supplied...

```
[Mime::EXTENSION_LOOKUP[params[:format]]]
```

```
module Mime
...
  ALL  = Type.new "*/*", :all
  TEXT = Type.new "text/plain", :text
  HTML = Type.new "text/html",:html, %w( application/xhtml+xml )
  XML  = Type.new "application/xml", :xml, %w( text/xml application/x-xml)
  ...
  SET = [ALL, TEXT, HTML, JS, ICS, XML, RSS, ATOM, YAML, JSON]

 # Encapsulates the notion of a mime type.
 class Type
   ...
   def initialize(string, symbol = nil, synonyms = [])
   ...
 end
  ...
  EXTENSION_LOOKUP = Hash.new { |h, k| h[k] = Type.new(k) unless k == "" }
  EXTENSION_LOOKUP["html"]  = HTML
  EXTENSION_LOOKUP["xhtml"] = HTML
  EXTENSION_LOOKUP["txt"]   = TEXT
  EXTENSION_LOOKUP["xml"]   = XML
  ...
end
```


49

Mime::Type is a convenience class that represents a MIME Type and provides easy access to the type name, extension and synonyms.

This slide does not show all the declarations for all pre-defined Mime::Type constants -- but they are all listed in the Array assigned to the SET constant.

Custom Types & Extensions

```
# in environment.rb
Mime::Type.register "easter_egg", :egg, %w(undocumented/whimsy)
```

```
class Mime::Type
  ...
  def register(string, symbol, synonyms = [])
    Mime.send :const_set, symbol.to_s.upcase,
              Type.new(string, symbol, synonyms)
    SET << Mime.send(:const_get, symbol.to_s.upcase)
    LOOKUP[string] = EXTENSION_LOOKUP[symbol.to_s] = SET.last
  end
  ...
end
```

```
EGG = Type.new("easter_egg", :egg, ["undocumented/whimsy"])
SET = [ALL, TEXT, HTML, JS, ICS, XML, RSS, ATOM, YAML, JSON, EGG]
LOOKUP["easter_egg"]=EGG
EXTENSION_LOOKUP["egg"]=EGG
```


50

If your action supports a type that is not included in the SET, you need to register it in environment.rb.

Mime::Type.register creates a Mime::Type constant for your custom type using the String, extension symbol and any synonyms you suppy. It then adds the new constant to the LOOKUP Hash and the EXTENSION_LOOKUP Hash and SET.

Determining Which MIME Type Gets Priority If params[:format] is not supplied...

```
@mime_type_priority =
  eval(
       "(params[:format] && Mime::EXTENSION_LOOKUP[params[:format]]) ? " +
       "[Mime::EXTENSION_LOOKUP[params[:format]]] : request.accepts",
       block_binding
   )
```

```
class AbstractRequest
   ...
    def accepts
      @accepts ||=
        if @env['HTTP_ACCEPT'].to_s.strip.empty?
          [ content_type, Mime::ALL ]
        else
          Mime::Type.parse(@env['HTTP_ACCEPT'])
        end
    end
  ...
end
```

```
# typical Firefox Accept header
text/xml,application/xml, application/xhtml+xml,
text/html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
```


51

If there's not a format value in the params Hash, Rails defers to request.accepts, which sets @mime_type_priority to Mime::ALL (the constant that corresponds to specifying "*/*" in the Accept header) if there's no Accept header associated with the request and calls Mime::Type.parse otherwise, passing it the Accept header.

Inside the oval is an example of an Accept header. It's the default header used when you browse with Firefox. The q's represent the q factor or quality factor, which is used to specify a percentage of the ideal quality that would be acceptable for a particular format. Sometimes it makes sense to sacrifice image or audio quality for increased download speed. If no q factor is supplied, a quality factor of "1" is assumed. For the remainder of this walk-through, lets assume the user made the request for details about a train station while browsing with Firefox.

Prioritize Supported Types Per the HTTP Spec

```ruby
def parse(accept_header)
  index = 0
  list = accept_header.split(/,/).map! do |i|
    AcceptItem.new(index += 1, *i.split(/;\s*q=/))
  end.sort!
  ...# Handle text\xml by replacing it with app\xml if necessary
  ...# Sort more specific xml-based types ahead of app/xml
  list.map! { |i| Mime::Type.lookup(i.name) }.uniq!
  list
end
```

```ruby
class AcceptItem
  ...
  def initialize(order, name, q=nil)
   ...
    q ||= 0.0 if @name == "*/*"
    @q = ((q || 1.0).to_f * 100).to_i
  end
  def <=>(item)
    result = item.q <=> q
    result = order <=> item.order if result == 0
    result
  end
  ...
end
```

52

Mime::Type.parse breaks down the Accept header into one or more AcceptItems depending on how many types are included in the Accept header.

AcceptItem is a helper class that sorts the specified types (by virtue of its custom spaceship operator definition) based q factor (quality factor) and order of appearance in the Accept header.

Mime::Type.parse returns an ordered Array of Mime::Type constants. Types with a higher q factor and that appear earlier in the Accept header end up at the top of the list.

Prioritize Supported Types Per the HTTP Spec

```
def parse(accept_header)
  index = 0
  list = accept_header.split(/,/).map! do |i|
    AcceptItem.new(index += 1, *i.split(/;\s*q=/))
  end.sort!
  ...# Handle text\xml by replacing it with app\xml if necessary
  ...# Sort more specific xml-based types ahead of app/xml
  list.map! { |i| Mime::Type.lookup(i.name) }.uniq!
  list
end
```

```
# Firefox
text/xml,application/xml,
application/xhtml+xml,
text/html;q=0.9,
text/plain;q=0.8,image/png,
*/*;q=0.5
```

```
class AcceptItem
  ...
  def initialize(order, name, q=nil)
   ...
    q ||= 0.0 if @name == "*/*"
    @q = ((q || 1.0).to_f * 100).to_i
  end
  def <=>(item)
    result = item.q <=> q
    result = order <=> item.order if result == 0
    result
  end
  ...
end
```

```
Mime::HTML
Mime::XML
Mime::PNG
Mime::TEXT
Mime::ALL
```


Given the default Firefox Accept header, Mime::Type.parse returns [HTML, XML, PNG, TEXT, ALL].

The `respond_to` Source

```
def respond_to(*types, &block)
  raise ArgumentError, "respond_to takes either types or a block..."
    unless types.any? ^ block

  block ||= lambda {|responder|types.each {|type| responder.send(type)}}

  responder = Responder.new(block.binding)

  block.call(responder)

  responder.respond
end
```

from ActionController::MimeResponds::InstanceMethods

54

Now we’re up to the penultimate line of code in respond_to. Here Rails actually calls the logic in the block we passed to respond_to (or the block that was created based on the supplied *types), passing it the newly-created Responder.

The `respond_to` Block Methods: MIA

```
block.call(responder)
```

```
respond_to do |format|
  format.html  <-
  format.xml {:xml => @train_station.to_xml}
  format.egg { render_and_stream_train }
end
```

HTML: METHOD MISSING

```
class Responder
  ...
  def method_missing(symbol, &block)
    mime_constant = symbol.to_s.upcase

    if Mime::SET.include?(Mime.const_get(mime_constant))
      custom(Mime.const_get(mime_constant), &block)
    else
      super
    end
  end
  ...
end
```


55

The block we passed to respond_to call from the show action we've been looking at throughout this presentation is at the top of this slide, for reference. First a method named "html" is called on the object passed to the block, which in this case is our newly-minted Responder.

There is no "html" method defined for our Responder. When an unknown method is called on an object, Ruby invokes method_missing for that object. The default behavior for method_missing is to raise an Error. But method_missing is overridden for the Responder class to call a method called "custom", passing it the Mime::Type constant that corresponds to the method name and the block that was passed to the method (if a block was passed to the method).

Link Supported Types & Responses: `format.html`

```
#called by Responder.method_missing
def custom(mime_type, &block)
  mime_type =
    mime_type.is_a?(Mime::Type) ?
    mime_type : Mime::Type.lookup(mime_type.to_s)

  @order << mime_type

  if block_given?
    @responses[mime_type] = Proc.new do
    eval "response.content_type = '#{mime_type.to_s}'",
      @block_binding
      block.call
    end
  else
    if source = DEFAULT_BLOCKS[mime_type.to_sym]
      @responses[mime_type] = eval(source,@block_binding)
    else
      raise ActionController::RenderError,
          "Expected a block but none was given ..."
    end
  end
end
```


56

The “custom” method adds the specified Mime::Type to its @order Array. Recall that the @order Array represents the response formats that the Controller action supports.

If a block is passed to custom (the Kernel method block_given? will return true), it means that a custom behavior block was provided for the format in the respond_to block, and a key\value pair will be added to the @responses Hash where the key is a Mime::Type constant and the value is a Proc based on that custom behavior block. The Proc also sets the content_type for the response based on the Mime::Type.

In this case, for HTML, a custom logic block was not provided, so Rails checks the DEFAULT_BLOCKS Hash to see if there’s a default block associated with HTML.

Default Handling for Standard Templates

```
class Responder
  DEFAULT_BLOCKS = [:html, :js, :xml].inject({}) do |blocks, ext|
    template_extension = (ext == :html ? '' : ".r#{ext}")
    blocks.update ext =>
      %(Proc.new { render :action => "\#{action_name}#{template_extension}",
      :content_type => Mime::#{ext.to_s.upcase} })
  end
  ...
end
```

```
DEFAULT_BLOCKS =
 {:xml=>
   "Proc.new { render :action => \"\#{action_name}.rxml\" ,
                      :content_type => Mime::XML }",
 :html=>
   "Proc.new { render :action => \"\#{action_name}\",
                      :content_type => Mime::HTML }",
 :js=>
   "Proc.new { render :action => \"\#{action_name}.rjs\",
                         :content_type => Mime::JS }"}
```


57

This slides shows how the DEFAULT_BLOCKS Hash gets populated. Rails loops through the default formats, adding a key\value pair to the Hash for each format. The key is the symbol for the format, and the value is a Proc based on a block that renders the View template the corresponds with the format.

Since there’s a DEFAULT_BLOCK associated with HTML...

Link Supported Types & Responses: `format.html`

```
#called by Responder.method_missing
def custom(mime_type, &block)
  mime_type =
    mime_type.is_a?(Mime::Type) ?
    mime_type : Mime::Type.lookup(mime_type.to_s)

  @order << mime_type

  if block_given?
    @responses[mime_type] = Proc.new do
    eval "response.content_type = '#{mime_type.to_s}'",
      @block_binding
      block.call
    end
  else
    if source = DEFAULT_BLOCKS[mime_type.to_sym]
      @responses[mime_type] = eval(source,@block_binding)
    else
      raise ActionController::RenderError,
          "Expected a block but none was given...     "
    end
  end
end
```

```
@order =[Mime::HTML]
```

```
@responses[Mime::HTML] =
  Proc.new { render :action => "show",
                     :content_type => Mime::HTML }
```


...a Proc based on the default logic for HTML is added to the @responses Hash.

Mime::HTML is added to @order (which represents the format supported by the Controller action).

The respond_to Block Methods: MIA

```
block.call(responder)
```

```
 respond_to do |format|
   format.html
   format.xml {:xml => @train_station.to_xml}
   format.egg { render_and_stream_train }
end
```

XML: METHOD MISSING

```
class Responder
  ...
  def method_missing(symbol, &block)
    mime_constant = symbol.to_s.upcase

    if Mime::SET.include?(Mime.const_get(mime_constant))
      custom(Mime.const_get(mime_constant), &block)
    else
      super
    end
  end
  ...
end
```


59

Now we return to executing the logic in the respond_to block. A method called “xml” is called on our Responder. Again, method_missing is invoked because there is no “xml” method defined for our Responder -- and again method_missing invokes the “custom” method.

Link Supported Types & Responses: `format.xml`

```
#called by Responder.method_missing
def custom(mime_type, &block)
  mime_type =
    mime_type.is_a?(Mime::Type) ?
    mime_type : Mime::Type.lookup(mime_type.to_s)

  @order << mime_type

  if block_given?
    @responses[mime_type] = Proc.new do
    eval "response.content_type = '#{mime_type.to_s}'",
      @block_binding
      block.call
    end
  else
    i
    
    e

    e
  end
end
```

```
@order=[Mime::HTML, Mime::XML]
```

```
@responses[Mime::XML] =
 Proc.new do
   eval "response.content_type = 'application/xml'",
   @block_binding
   block.call
 end
```

```
{render :xml =>@train_station.to_xml}
```


This time through "custom", we added XML to the @order Array (which represents the formats the Controller action supports). We also add a key\value pair to the @responses Hash, where the key is Mime::XML and the value is a Proc based on the custom logic block paired with XML in the respond_to block.

The `respond_to` Block Methods: MIA

```
block.call(responder)
```

```
 respond_to do |format|
   format.html
   format.xml {:xml => @train_station.to_xml}
   format.egg { render_and_stream_train }
end
```

EGG: METHOD MISSING

```
class Responder
  ...
  def method_missing(symbol, &block)
    mime_constant = symbol.to_s.upcase

    if Mime::SET.include?(Mime.const_get(mime_constant))
      custom(Mime.const_get(mime_constant), &block)
    else
      super
    end
  end
  ...
end
```


61

Now we're up to the last line in the respond_to block. We call a method named "egg" on the Responder, which triggers method_missing, which invokes "custom".

Link Supported Types & Responses: `format.egg`

```
#called by Responder.method_missing
def custom(mime_type, &block)
  mime_type =
    mime_type.is_a?(Mime::Type) ?
    mime_type : Mime::Type.lookup(mime_type.to_s)

  @order << mime_type

  if block_given?
    @responses[mime_type] = Proc.new do
    eval "response.content_type = '#{mime_type.to_s}'",
      @block_binding
      block.call
    end
  else
    i
    e
    e
  end
end
```

```
@order=
[Mime::HTML,Mime::XML,Mime::EGG]
```

```
@responses[Mime::EGG] =
 Proc.new do
   eval "response.content_type = 'easter_egg'",
   @block_binding
   block.call
 end
```

```
{render_and_stream_train}
```


This time through "custom" we add Mime::EGG to @order. The @responses Hash gets a key\value pair where the key is Mime::EGG and the value is a Proc based on the custom Easter egg logic.

The `respond_to` Source

```
def respond_to(*types, &block)
  raise ArgumentError, "respond_to takes either types or a block..."
    unless types.any? ^ block

  block ||= lambda {|responder|types.each {|type| responder.send(type)}}

  responder = Responder.new(block.binding)

  block.call(responder)

  responder.respond
end
```

from `ActionController::MimeResponds::InstanceMethods`

Now we're up to the last line of respond_to. It's where Rails actually issues a response to the client. A method called "respond" is invoked on our Responder.

Are Our Priorities in @order?

```
def respond
  for priority in @mime_type_priority
    if priority == Mime::ALL
      @responses[@order.first].call
      return
    else
      if priority === @order
        @responses[priority].call
        return # mime type match found, be happy and return
      end
    end
  end
  # if :all is not supported, return an error status
end
```

```
# @mime_type_priority
Mime::HTML
Mime::XML
Mime::PNG
Mime::TEXT
Mime::ALL
```

```
# @order
[Mime::HTML, Mime::XML, Mime::EGG]
```


Here we loop through the types in the @mime_type_priority Array (which represents the formats the client will accept) until we find a match in the @order Array (which represents the formats the Controller action supports). If there had been a format value in the params Hash, @mime_type_priority would have been an Array with a single element -- the Mime::Type constant associated with that one format.

If the top priority is Mime::ALL, the logic linked to whatever format is referenced first in the respond_to block gets executed.

Are Our Priorities in @order?

```
def respond
  for priority in @mime_type_priority
    if priority == Mime::ALL
      @responses[@order.first].call
      return
    else
      if priority === @order
        @responses[priority].call
        return # mime type match found, be happy and return
      end
    end
  end
  # if :all is not supported, return an error status
end
```

```
# @mime_type_priority
Mime::HTML
Mime::XML
Mime::PNG
Mime::TEXT
Mime::PNG
```

```
# @order
[Mime::HTML, Mime::XML, Mime::EGG]
```

```
class Mime::Type
  def ===(list)
    if list.is_a?(Array)
      (@synonyms + [ self ]).any? { |synonym| list.include?(synonym) }
    else
      super
    end
  end
end
```


Otherwise, the threequal operator, which we talked about in the first part of the presentation, is used to determine whether there's a match. It is overridden for Mime::Type to compare a single Mime::Type with an Array of Mime::Types, and also to take the synonyms for the types into account.

Many Happy Returns

```
def respond
  for priority in @mime_type_priority
    if priority == Mime::ALL
      @responses[@order.first].call
      return
    else
      if priority === @order
        @responses[priority].call
        return # mime type match found, be happy and return
      end
    end
  end

  if @order.include?(Mime::ALL)
    @responses[Mime::ALL].call
  else
    eval 'render(:nothing => true,
          :status => "406 Not Acceptable")', @block_binding
  end
end
```

```
# @mime_type_priority
Mime::HTML
Mime::XML
Mime::PNG
Mime::TEXT
Mime::ALL
```

```
# @order
[Mime::HTML,Mime::XML,Mime::EGG
```

66

Rails responds to the client’s request by calling the Proc stored in @responses for first type in @mime_type_priority that also exists in @order.

Here we have a match! Rails determined that the client “wants” HTML and responds to the client’s request to show train station details on a Web page by rendering show.rthml.

The comment “be happy and return” actually exists in the Rails source!

Mozilla Firefox Start Page

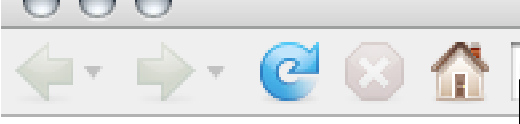
http://localhost:3000/tickets/1.egg_9_3_4

67

The actual presentation concluded with a demo. I didn't want to demo what happens when you tack ".egg" onto the end of the URL mapped to the show action for TrainStations. I felt the screenshot of the browser on one of the slides was sufficient for that.

So I brought up Harry Potter again. Students take the Hogwarts Express to school at the beginning of each school year. The train leaves from Platform 9 3/4. Harry is bewildered when he goes to the train station to journey to Hogwarts for the first time. He can't find Platform 9 3/4. Between Platform 9 and Platform 10, there's a brick barrier. He learns that the only way you can get to Platform 9 3/4 is by magically walking through that barrier.

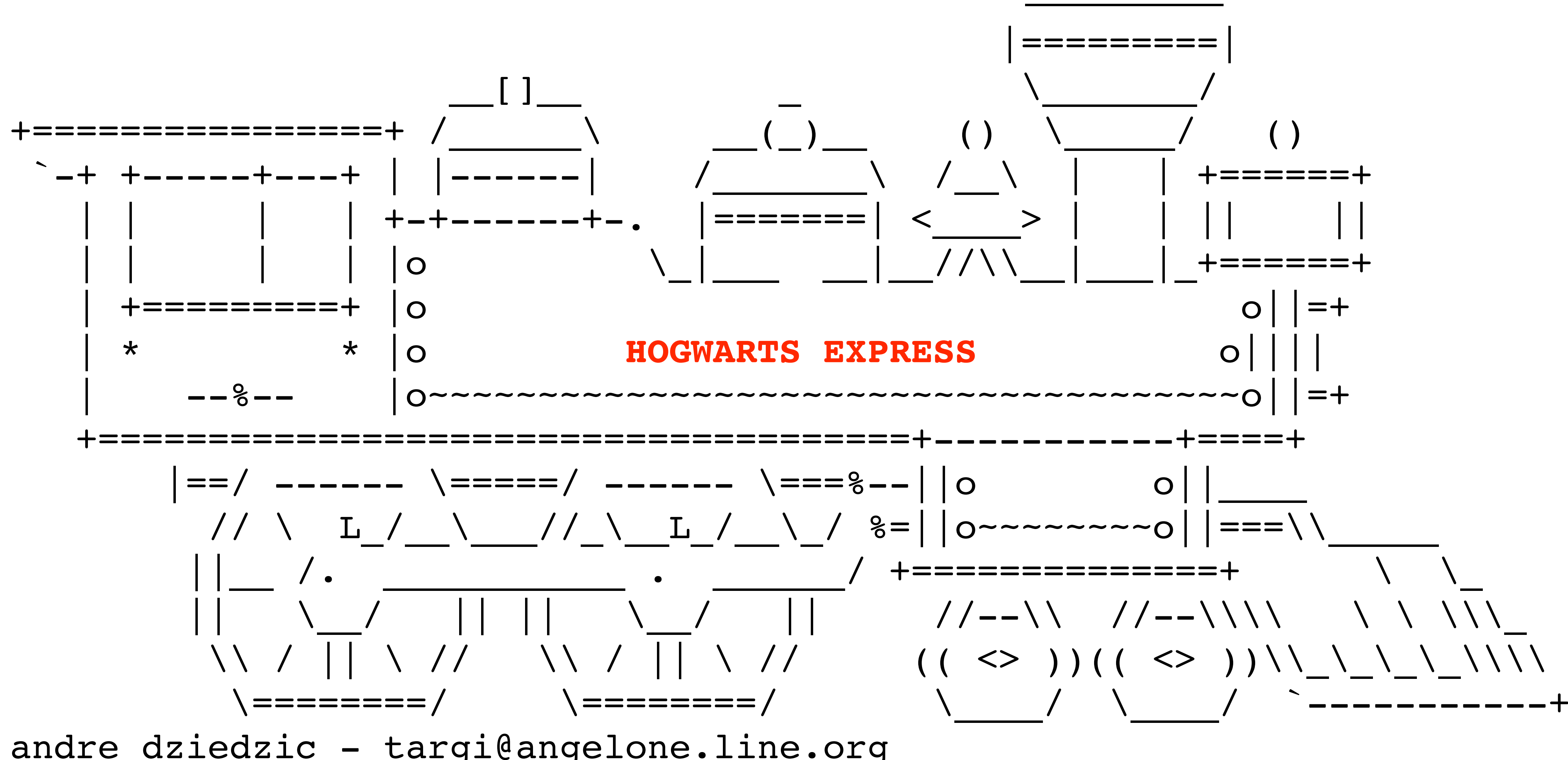
So..I typed the URL mapped to the show action with an egg_**9_3_4** extension....



Mozilla Firefox Start Page

http://localhost:3000/tickets/1.egg

http://localhost:3000/tickets/1.egg_9_3_4



ASCII art from http://www.ascii-art.de (sans 'Hogwarts Express')

68

...and this is what shows up.

How did I implement that? I leave that as an exercise for the reader!