
The Nitro and Og Todo List Tutorial

Arne Brasseur

This article may be freely distributed under the terms of the GNU Free Documentation License Version 1.2. The full text of the license can be found on-line at <http://www.gnu.org/licenses/fdl.html>.

Abstract

Build a simple web application from the ground up using the Ruby programming language and the Nitro web framework.

Table of Contents

Introduction	1
Getting and installing Nitro	1
hello, world	3
Building a real app	3
Building the model	4
Views : building templates	5
The Controller enters the stage	6
Getting interactive	7
Opening and closing tasks	8
Elements : custom HTML tags	8
Prettify	10

Introduction

Nitro is a framework to build dynamic web applications using the lightweight Ruby programming language. The sister project Og, short for Object-Graph, can be used to access a database.

There are many web frameworks already, what sets Nitro apart?

Nitro is, like Ruby, lightweight. Nitro applications as well as the Nitro code itself read like standard Ruby and are easy to understand. Standard Ruby idioms are preferred over exotic and inaccessible shorthand notation. It is faster than several competitors, and serves pages in two steps : template compilation and template rendering. Compilation happens only the first time a page is served, amortizing the overhead.

Getting and installing Nitro

Installing Nitro/Og can be as simple as `gem install nitro og`. This will pull in some other projects. Much of the web application framework functionality resides in the subproject Raw which will be pulled in together with Nitro and Og.

There are a number of other dependencies:

Nitro/Og dependencies

Facets A large collection of extensions to Ruby its core libraries, and other reusable parts that could've been part of Ruby's core.

English	A collection of natural language processing and text manipulation methods.
Blow	Web libraries, include markup mixins, builders and microformat libraries.
Opod	Opod stands for Object-pods, it takes care of persisting arbitrary Ruby objects. It is used by some types of session store.
Xml-Simple	A XML reading/writing API that offers more convenience than the raw REXML.
UUIDtools	UUIDs are globally unique identifiers. This is needed to run the blog example that comes with Nitro.
TMail	This is also needed for the blog example. TMail is an email handler library for Ruby. TMail can extract data from mail, and write data to mail following the relevant RFCs on the subject.
Mongrel	A Ruby webserver. There are other options such as WEBrick or FastCGI, but Mongrel is recommended.
RedCloth	A markup library. There are some macros that can be used in templates that will call the markup code. By default these use RedCloth, although you can use a different markup processor if you prefer.

When installing the official gems these should be installed automatically. If you're making your own gems from the repository code you'll have to install them by hand:

```
gem install facets
gem install english
gem install blow
gem install opod
gem install xml-simple
```

For the blog example:

```
gem install uuidtools
gem install tmail
```

Recommended:

```
gem install mongrel
gem install redcloth
```

The latest development source can be retrieved using the version control system Darcs :

```
darcs get http://repo.nitroproject.org
```

To stay up to date, issue 'darcs pull' in the Nitro root directory. Once you got hold of the sources it is possible to build your own gems with rake:

```
cd repo.nitroproject.org
rake dist:all
cd dist
gem install glue
gem install og
gem install raw
gem install nitro
```

If you do not wish to install unofficial gems, you can use the repository version directly. You will need to add the directory `nitro/bin` to your `PATH` to use the `nitro` command. The file `script/lib/glycerin.rb` can take care of setting your Ruby loadpath to include the various Nitro subprojects. For instance:

```
export NITRO_HOME=/path/to/repo.nitroproject.org
export PATH=$NITRO_HOME/nitro/bin:$PATH
export RUBYOPT="-rubygems -I$NITRO_HOME/script -rlib/glycerin"
```

hello, world

Citing Brian Kernighan. A Nitro application can be as simple as

```
require "nitro"
include Nitro

class RootController
  def index
    @out << "hello, world"
  end
end

Nitro.start(RootController)
```

Save this as `app.rb`.

Note

At the time of writing it won't work without a file named `conf/debug.rb`. For now it will suffice to put this in there:

```
def setup(app)
end
```

Building a real app

To show off how easy it is to get started with Nitro we'll build a `ToDoList` web application. To get started quickly it is possible to let Nitro generate a base application to start from. The syntax is:

```
nitro create todolist
```

With Nitro the programmer is free to structure the application anyway she chooses. There are only a few directories and filenames that have a special meaning to Nitro, and even these can be changed from their default values. Pretty much everything is optional : you don't need a directory for your models, templates or unit tests until you actually need it. 'create', however shows the recommended way to structure things. By sticking to this layout it is easy for others (or yourself) to find your way around.

`app.rb` is your main Nitro application, the **nitro** command will look for this file when starting the app.

The `app/` subdirectory contains `model/`, `template/` and `controller/` subdirectories for the three parts of the MVC pattern. Model and controller files have to explicitly included from `app.rb`, and hence can be placed anywhere. The `app/template` location is the default location for templates.

The `conf/` subdirectory contains a `debug.rb` and a `live.rb` file, here custom configuration can be done. The appropriate file will be loaded depending on the mode nitro runs in.

Building the model

In order to store the tasks on our todo list we need to tell Og where to find the database. The Og start method does the trick :

```
Og.start(:adapter => :sqlite)
```

or

```
Og.start(
  :adapter => :mysql,
  :user => "root",
  :password => "password"
)
```

This goes in `conf/debug.rb` inside the `setup(app)` method. We also need to change `app.rb` to load Og, add this line at the top:

```
require "og"
```

In a normal Ruby application you might have this class :

```
class TodoItem
  attr_accessor :title
  attr_accessor :done
end
```

But this is a web app, that would be too easy, right?

Create a file named `app/model/todoitem.rb` :

```
class TodoItem
  attr_accessor :title, String
  attr_accessor :done, TrueClass
end
```

And require that file from `app.rb`. As you can see we told Og the type of our attributes, string and boolean. That way it knows what type to give to each column in the database schema. Running the app we can see in our terminal

```
INFO: Og uses the Sqlite store.
DEBUG: Og manageable classes: [TodoItem]
DEBUG: CREATE TABLE ogtodoitem (title text, done boolean, oid integer PRIMARY KEY)
```

Nice.

You can play around a bit with your model using Nitro's console mode. Create a few items for the todo list, so we have something to show later on.

```
$ nitro console
INFO: Og uses the Sqlite store.
DEBUG: Og manageable classes: [TodoItem]
DEBUG: CREATE TABLE ogtodoitem (title text, done boolean,
  oid integer PRIMARY KEY)
DEBUG: SELECT * FROM ogtodoitem LIMIT 1
INFO: Starting Script server in debug mode, listening at
  0.0.0.0:9000
```

```
INFO: This console is attached to the application context.
INFO:
INFO: * $app points to the application
INFO: * $srv points to the adapter
INFO: * use get(uri), post(uri), response() to
      programmatically call actions
INFO:
irb> TodoItem.all
=> []

irb> TodoItem.create_with(:title => "Walk the dog",
  :done => false)
=> #<TodoItem:0xb77d00e8 @done=false,
  @title="Walk the dog", @oid=1, @validation_errors={}>

irb> TodoItem.create_with(:title => "Water the plants",
  :done => false)
=> #<TodoItem:0xb77b7cc8 @done=false,
  @title="Water the plants", @oid=2, @validation_errors={}>

irb> TodoItem.all
=> [#<TodoItem:0xb77b3380 @done=false, @title="Walk the dog", @oid=1>,
  #<TodoItem:0xb77b1f44 @done=false, @title="Water the plants", @oid=2>]

irb> TodoItem[2].title
=> "Water the plants"

> item=TodoItem.new
=> #<TodoItem:0xb77a7d64>

irb> item.title="Buy chunky bacon"
=> "Buy chunky bacon"

irb> item.done=true
=> true

irb> item.save
=> 1

irb> TodoItem.all
=> [#<TodoItem:0xb7799110 @done=false, @title="Walk the dog",
  @oid=1>, #<TodoItem:0xb7797964 @done=false, @title="Water
  the plants", @oid=2>, #<TodoItem:0xb7796500 @done=true,
  @title="Buy chunky bacon", @oid=3>]

irb> item.oid
=> 3
```

Views : building templates

When you launch the app (with 'nitro') and go to <http://127.0.0.1:9000> you see a welcome page with some information to get started. This file can be found in `app/template/index.html`. Let's change it into something useful:

```
<html>
  <head>
    <title>Todolist</title>
  </head>
  <body>
    <h1>TodoList</h1>
    <ul>
      <?r TodoItem.all.each do |i| ?>
        <li>#{i.title} - #{i.done}</li>
      <?r end ?>
    </ul>
  </body>
</html>
```

Let's clean this up a bit using the Nitro concept of morphers

```
<ul>
  <li for="i in TodoItem.all">#{i.title} - #{i.done}</li>
</ul>
```

Ahh, much better, but there's some stuff here that should go in the controller!

The Controller enters the stage

MVC consists of three parts : models manage your data, views display the data, and controllers handle user input. Controllers and views tend to get closely coupled.

In a web application every page load is handled by a controller 'action'. The controller will perform any necessary actions on the models, and load and prepare the data that is later displayed by the views (templates).

In Nitro a URL is typically interpreted as **/controller/action/parameters**. There is also a root controller to handle requests where there is no controller explicitly specified. When there is no action specified Nitro will render the **'index'** action.

Create the file `app/controller/root.rb` as follows:

```
class RootController
  def index
    @items = TodoItem.all
  end
end
```

We also have to adapt `app.rb`. We need to require the file, and set our new class as root controller. This is what `app.rb` looks like afterwards :

```
require "nitro"
require "og"
include Nitro

require "app/model/todoitem"
require "app/controller/root"

Nitro.start(RootController)
```

Now when surfing to `http://127.0.0.1:9000`, the `index` method of the `root` controller is called, and afterwards the template is rendered. Since the controller loads the model items now, we can change the template:

```
<li for="i in @items">#{i.title} - #{i.done}</li>
```

Getting interactive

If we're only going to display a list of tasks, we might as well use some static HTML. Next up we'll let the user add new tasks. We'll create a controller action `new` on our root controller that shows a form, which will be submitted to the `create` action.

At the beginning of the page under the `<h1>...</h1>` we'll add this link:

```
<a href="#{R :new}">New task</a>
```

The `R` method is a helper for creating URI's. It can be found in the module `Raw::EncodeURI`. Its usage is documented in the RDoc. Here we simply pass the name of an action on our current controller. Since this is the root controller, this will result in the URI `'/new'`.

Look at the front page again, you should see the link. When you click it, however, Nitro presents an error page:

```
Internal Server Error
Path: /new
wrong number of arguments (1 for 0)
```

The problem is we haven't yet created the `'new'` action. Since there's no controller mounted at `'/new'`, the root controller is used. This controller doesn't have a `'new'` action so the default `index` action is used. The remainder of the URI is passed as arguments to the action method. This method isn't expecting any arguments, hence the error.

So, create a file `app/template/new.html` :

```
<html>
  <head>
    <title>Todolist</title>
  </head>
  <body>
    <h1>Add new task</h1>
    #{form(:object => TodoItem, :action => :create, :method => :post) do |f|
      f.attribute :title
      f.attribute :done
      f.submit 'Create'
    end}
  </body>
</html>
```

Here we use the `FormHelper` to create a form. We tell it it's a form for a `TodoItem`, so it's smart enough to figure out the title is a string and 'done' is boolean. The result is a form with a textbox and a checkbox.

First however we have to make this form method available to our template, add this to the controller :

```
include FormHelper
```

And while you're at it, also add the method that will handle the form:

```
def create
  request.assign(TodoItem.new).save
  redirect_to :index
end
```

Opening and closing tasks

All that is left is being able to mark tasks as done. Let's add a method to our controller to do just that:

```
def toggle(oid)
  item = TodoItem[oid]
  item.done = !item.done
  item.save
  redirect_referrer
end
```

This method takes one parameter, which will be taken from the URI, for instance **http://127.0.0.1:9000/toggle/3**. It gets an item from the database, changes its state and saves it again. Afterwards it points the browser back to whichever page it came from.

In **index.html** we'll add a link to every item in our list:

```
<li for="item in @items">#{item.title} - #{item.done} -
  [<a href="#{R :toggle}/#{item.oid}">Toggle</a>]</li>
```

That's all it takes, reload the page and see if it works.

Another feature we'll need is deleting tasks altogether. This follows roughly the same pattern.

A controller action:

```
def delete(oid)
  TodoItem.delete oid
  redirect_referrer
end
```

And a small adaptation of our template:

```
<li for="item in @items">#{item.title} - #{item.done} -
  [<a href="#{R :toggle}/#{item.oid}">Toggle</a>]
  [<a href="#{R :delete}/#{item.oid}">Delete</a>]
</li>
```

Elements : custom HTML tags

Here is our **index.html** thus far:

```
<html>
  <head>
    <title>Todo list</title>
  </head>
  <body>
    <h1>Todo list</h1>
    <a href="#{R :new}">New task</a>
```

```
<ul>
  <li for="item in @items">#{item.title} - #{item.done} -
    [<a href="#{R :toggle}/#{item.oid}">Toggle</a>]
    [<a href="#{R :delete}/#{item.oid}">Delete</a>]
  </li>
</ul>
</body>
</html>
```

How about we change this to something a little more elegant?

```
<Page title="Todo list">
  <a href="#{R :new}">New task</a>
  <ul>
    <li for="i in @items">
      <Task item="i" />
    </li>
  </ul>
</Page>
```

And this is how `app/template/new.html` ends up looking:

```
<Page title="Add new task">
  #{form :object => TodoItem, :action => :create, :method => :post do |f|
    f.attribute :title
    f.attribute :done
    f.submit "Create!"
  end}
</Page>
```

To achieve this we'll create two special classes, **Page** and **Task**. When these tags are encountered in the template, an instance of the corresponding class is instantiated. The attributes in the template are made available as instance variables. Eventually the special method **render** is called. The result is placed in the template instead of the original tags.

These 'elements' act as template macros, they allow you to rewrite a part of your template. Whatever is inside the tags is available to **render** by calling **content**.

So without further ado, start by creating a directory for your elements named `app/element`.

This is what the file `app/element/page.rb` looks like:

```
class Page
  def render
    %~<html>
      <head>
        <title>#{@title}</title>
      </head>
      <body>
        <h1>#{@title}</h1>
        #{content}
      </body>
    </html>~
  end
end
```

This is pretty straightforward, the `@title` instance variable is set to whatever you filled in in the template, the `content` method returns whatever is found inside the `Page` tags.

The `Task` class gets a little trickier, let's have a closer look:

```
class Task
  def item_attr(at)
    "\#{#{@item}.\#{at}}"
  end

  def render
    %~\#{item_attr :title} - \#{item_attr :done}
      [<a href="\#{R :toggle}/\#{item_attr :oid}">Toggle</a>]
      [<a href="\#{R :delete}/\#{item_attr :oid}">Delete</a>]
    ~
  end
end
```

What is going on here? Remember Nitro processes templates in two steps, the first time a template is needed it is converted (compiled) into Ruby code, which is injected into the controller class as a special method. Afterwards the rendering of a template is simply a call to this method.

Elements are processed at compile time. So the `render` render method is called only once when the template is needed for the first time. Any code that needs to be evaluated every time a page is requested needs to be embedded in the String that is returned by `render`.

To achieve this we use a little trick. Remember that in a literal string you can write code in between `#{` and `}`. This code will be evaluated, and the result ends up as part of the string. We can use this to insert text at compile time.

If we want to defer that interpolation we have to put a literal `#{...}` in the the generated string, without it being interpolated. We can achieve this by 'escaping' it with a backslash : `"\#{like_this}"`. The result is that the string interpolation happens each time the template is rendered, it's as if it was written directly in the template.

In our template we had this little snippet:

```
<li for="i in @items">
  <Task item="i" />
</li>
```

The result is that in `Task#render` the variable `@item` is set to the string `"i"`. We happen to know that that string is the name of a local variable, so we return a string which looks like `\#{i.title} - \#{i.done}` - This will get evaluated later on.

The `item_attr` helper just makes it a little bit easier for ourselves.

We only scratched the surface of what is possible with elements. When you have nested elements (like `Task` inside `Page`) the inner elements are available through the variable `@_children`. This allows you to write families of elements that work together to create a complex DSL.

Prettify

- Turn it into a table
- add some CSS

- chapter : configuration
- chapter : aspects and annotations
- chapter : validation and error handling