
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	3
Views : building templates	5
The Controller enters the stage	6
Getting interactive	6

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 irb. Create a few items for the todo list, so we have something to show later on.

```
$ irb
irb> require "og"
irb> require "app/model/todoitem"
irb> Og.start(:adapter=>:sqlite)
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://localhost:9000> you see a welcome page with some information to get started. This file can be found in `app/template/index.htmlx` . 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://localhost: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`. It's 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
```