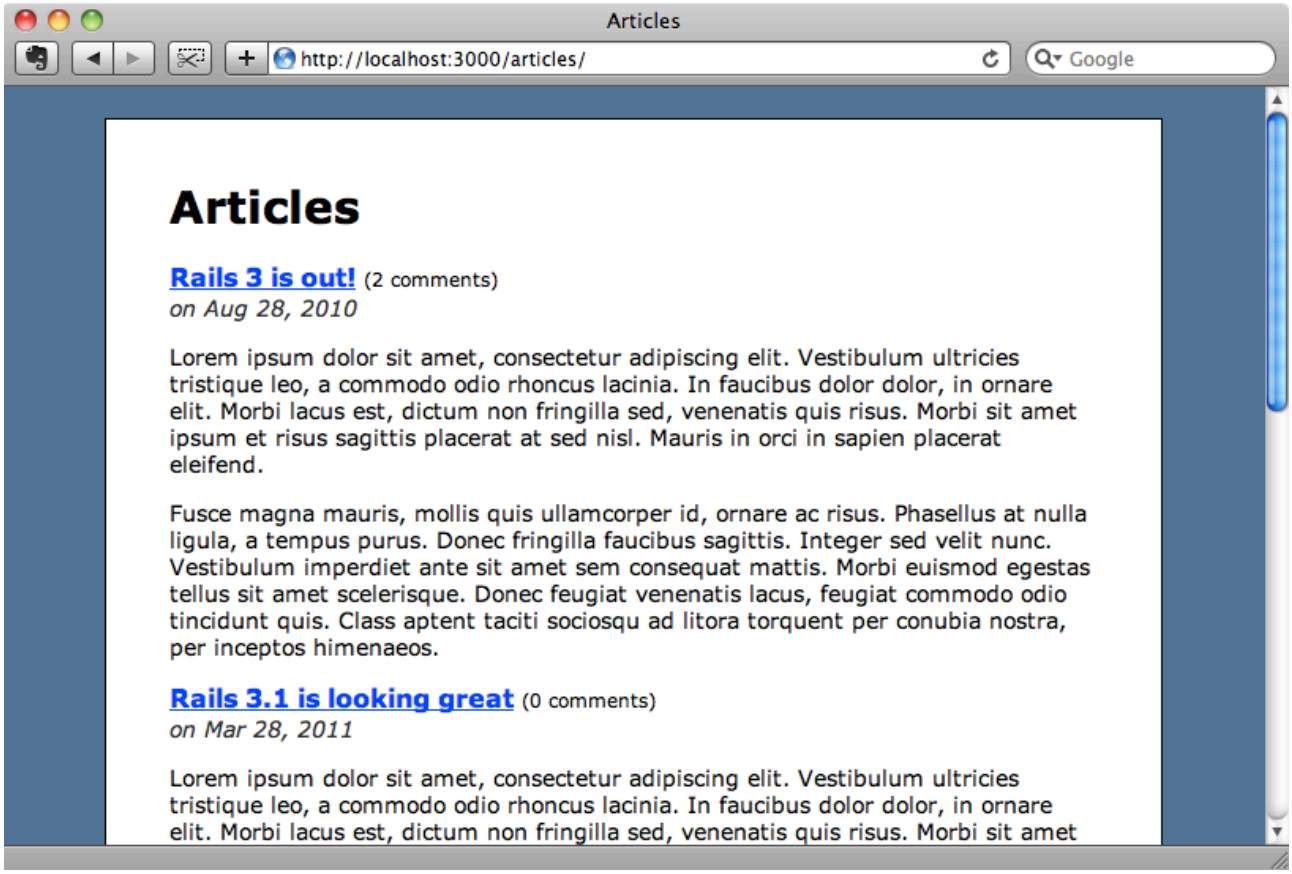
Episode 259

Decent Exposure

In this episode we're going to look at a gem called decent_exposure[1]. This is a simple gem with a nice concept. We can use it to create an interface of methods in the controller that the view can access instead of using instance variables. The gem uses a method called `expose` to define this interface.

Before we look at decent_exposure we'll try applying this concept manually. The application we'll be working with is a simple blogging application with many `Articles`, each of which can have many `Comments`.



The `ArticlesController` contains pretty standard controller code. For example in the `index` action it creates an instance variable called `@articles`.

[1] https://github.com/voxdolo/decent_exposure

/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.order(:name)
  end

  #Other actions omitted.
end
```

In the `index` view code we then use `@articles` to loop through each article and render it.

When you first start using Rails it may seem strange that access to a controller's instance variables is shared with the views as these are usually kept private within a class. We'll take a look now at an alternative approach that shares data by exposing methods to the views. To accomplish this we'll create private methods in the controllers that return either single models or lists of models.

First we'll remove the line of code that fetches the articles from the `index` action and put it into a method called `articles`. We want `@articles` to be used as a cache so that the articles are only fetched once. We can use the `||=` operator to achieve this and we won't now reference this instance variable anywhere else in the controllers or views. We'll make `articles` a helper method so that we can use it in the views.

/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  def index
  end

  private
  def articles
    @articles ||= Article.order(:name)
  end
  helper_method :articles
end
```

In the view we can now replace the call to the instance variable with a call to our new `articles` method.

/app/views/articles/index.html.erb

```
<% title "Articles" %>

<div id="articles">
<% for article in articles %>
  <h2>
    <%= link_to article.name, article %>
    <span class="comments">(<%= pluralize(article.comments.size,↵
      'comment') %>)</span>
  </h2>
  <div class="created_at">on <%= ↵
    article.created_at.strftime('%b %d, %Y') %></div>
  <div class="content"><%= simple_format(article.content) %></div>
<% end %>
</div>

<p><%= link_to "New Article", new_article_path %></p>
```

The other actions in the controller either find or create a single article and we can do something similar for those by creating an `article` method. This will be a little more complex than `articles` as it needs to do different things depending on the parameters that are passed in. Here's what the method looks like:

/app/controllers/articles_controller.rb

```
def article
  @article ||= params[:id] ? Article.find(params[:id]) : ↵
    Article.new(params[:id])
end
helper_method :article
```

If an `id` parameter exists then the method will find the `Article` with that `id`. If not then it will create a new one based on any `article` parameters. We now have a method that we can use to replace the `@article` instance variable and so we can remove the line in each action that finds or creates an action.

/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  def index
  end

  def show
  end

  def new
  end

  def create
    if article.save
      redirect_to articles_path, :notice => "Successfully ↵
        created article."
    else
      render :new
    end
  end

  def edit
  end

  def update
    if article.update_attributes(params[:articles])
      redirect_to articles_path, :notice => "Successfully ↵
        updated article."
    else
      render :edit
    end
  end

  def destroy
    @article.destroy
    redirect_to articles_url, :notice => "Successfully ↵
      destroyed article."
  end
```

/app/controllers/articles_controller.rb

```
private
  def articles
    @articles ||= Article.order(:name)
  end
  helper_method :articles

  def article
    @article ||= params[:id] ? Article.find(params[:id]) ↵
      : Article.new(params[:id])
  end
  helper_method :article
end
```

Some of our actions now have no code in them at all as they didn't have any behaviour other than to define an instance variable and this is now handled by the `action` method. We still need to go into each of the `ArticleController`'s views and replace each use of an instance variable with a call to the appropriate method. For example, the `show` view will look like this after it has been modified:

/app/views/articles/show.html.erb

```
<% title article.name %>

<%= simple_format article.content %>

<p>
  <%= link_to pluralize(article.comments.size, 'Comment'), ↵
    [article, :comments]%> |
  <%= link_to "Back to Articles", articles_path %> |
  <%= link_to "Edit", edit_article_path(article) %> |
  <%= link_to "Destroy", article, :method => :delete, ↵
    :confirm => "Are you sure?" %>
</p>
```

We won't show the other views but we'll need to change them in a similar way.

Another advantage of this approach is that it gives us lazy loading. If we were add action caching to, say, the `show` action, then the article that is being shown will only be fetched from the database if we're going to render the view; the article isn't

required in the controller layer. Action caching works well here as the action isn't fetched unless the controller truly needs it.

Adding The decent_exposure Gem

We have a good solution now but it would be even better if there was a more convenient way to define the methods that we expose to the view and this is where decent_exposure comes in. We can use its `expose` method which will define methods that expose models to the view in a similar way to the `articles` and `article` methods we wrote above. The `expose` method has some useful defaults: it will search for a model by its `id` parameter and if that isn't available it will build a new model using any appropriate parameters it can find. This means that we can use the default values for finding or creating single models. If we require any other behaviour we can pass a block to the method and define it there. Caching will be handled automatically for us by decent_exposure.

Let's try using it in our application. First we'll need to add the gem to the `Gemfile` and run the `bundle` command.

`/Gemfile`

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'
gem 'decent_exposure'
```

We can now replace the `article` and `articles` methods we wrote in the `ArticlesController` with two calls to `expose`.

/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController

  expose(:article)
  expose(:articles) { Article.order(:name) }

  def index
  end

  # Other actions omitted
end
```

The defaults that `expose` uses are correct for a single `Article` but if we want a list of articles we need to supply the custom behaviour to get those so we've copied it from the `articles` method and placed it in `expose`'s block.

If we reload our application now it'll work just as it did before, but the controllers are much neater as they're using the methods supplied by decent_exposure instead of instance variables.

Handling Nested Resources

How does decent_exposure handle nested `resources` such as the comments in our application that are nested under `articles`?

/config/routes.rb

```
Blog::Application.routes.draw do
  root :to => "articles#index"

  resources :articles do
    resources :comments
  end
end
```

Here's how the `CommentsController` looks:

/app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController
  def index
    @article = article.find(params[:article_id])
    @comments = @article.comments
    @comment = Comment.new
  end

  def new
    @article = Article.find(params[:article_id])
    @comment = @article.comments.build
  end

  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.build(params[:comment])
    if @comment.save
      redirect_to @comment.article, :notice => "Successfully ↵
        created comment!"
    else
      render :new
    end
  end
end
```

We're still using instance variables here. At the top of each action we fetch an article and then fetch or build a comment through that article. We can use decent_exposure here as it fully supports nested resources.

As before we'll replace each instance variable with a call to `expose`. We can use the default behaviour for fetching a single `Article` and `Comment` but to fetch the list of comments we'll need some custom behaviour. We can then delete the lines of code in the controller that assign instance variables and in the code that remains replace those instance variables with calls to the appropriate methods. Once we've made these changes the controller looks a lot cleaner:

/app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController
  expose(:article)
  expose(:comments) { article.comments }
  expose(:comment)

  def index
  end

  def new
  end

  def create
    if comment.save
      redirect_to comment.article, :notice => ↵
        "Successfully created comment!"
    else
      render :new
    end
  end
end
```

As we did when we changed the `ArticlesController` we'll need to update the views related to this controller so that they call the methods generated by decent_exposure instead of the instance variables, such as in the form partial below.

/app/views/comments/_form.html.erb

```
<%= form_for [article, comment] do |f| %>
  <%= f.error_messages %>
  <%= f.hidden_field :article_id %>
  <p>
    <%= f.label :name %>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :content, "Comment" %><br />
    <%= f.text_area :content, :rows => 12, :cols => 35 %>
  </p>
  <p><%= f.submit %></p>
<% end %>
```

When we try the application now it still works as well as it did earlier but the code in the controllers is much improved.

There's one potential gotcha with decent_exposure that we need to be aware of. When we use `expose` with its default behaviour to fetch as single model it will look for a pluralized version of the name that is passed in (e.g. `:article`) and attempt to fetch and build records through that scope if it's available. For example, let's say that we have the following two calls to `expose` in the `ArticlesController`.

/app/controllers/articles_controller.rb

```
expose(:article)
expose(:articles) { Article.order(:name).where(:visible => true) }
```

Any call to the singular `article` method will try to fetch an article based on a plural `articles` scope so, given the code, above when we look for a single article it will only be returned if that article is `visible`. If we don't want this behaviour then we'll need to change the name of the pluralized version so that it has a more descriptive name. In this case we'll rename it to `visible_articles`.

/app/controllers/articles_controller.rb

```
expose(:article)
expose(:visible_articles) { Article.order(:name).where(:visible ↵
  => true) }
```

Now the second `expose` call is no longer considered the base default scope that member actions are built on. Once we've made a change like this we will of course need to change any calls to that method in the views.

Changing The Default Behaviour

If we ever need to change the `expose` method's default behaviour we can do so by making a call to `default_exposure` and passing it a block. The behaviour we define in that block will override the default behaviour. The name passed to `expose` will be passed to the block in `default_exposure`.

```
class MyController < ApplicationController
  default_exposure do |name|
    ObjectCache.load(name.to_s)
  end
end
```

We generally won't need to override the default but the option is there if we ever need to.

That's it for this episode. It's a neat solution for cleaning up controllers and well worth considering if you think it'll fit in with your workflow.