Rails Routing from the Outside In

Ruby on Rails

Rails Routing from the Outside In

Rails Routing from the Outside In

This guide covers the user-facing features of Rails routing. After reading this guide, you will know:

- How to interpret the code in routes.rb.
- How to construct your own routes, using either the preferred resourceful style or the match method.
- What parameters to expect an action to receive.
- How to automatically create paths and URLs using route helpers.
- Advanced techniques such as constraints and Rack endpoints.

1 The Purpose of the Rails Router

The Rails router recognizes URLs and dispatches them to a controller's action. It can also generate paths and URLs, avoiding the need to hardcode strings in your views.

1.1 Connecting URLs to Code

When your Rails application receives an incoming request for:

```
GET /patients/17
```

it asks the router to match it to a controller action. If the first matching route is:

```
get '/patients/:id', to: 'patients#show'
```

the request is dispatched to the `patients` controller's `show` action with `{ id: '17' }` in `params`.

1.2 Generating Paths and URLs from Code

You can also generate paths and URLs. If the route above is modified to be:

```
get '/patients/:id', to: 'patients#show', as: 'patient'
```

and your application contains this code in the controller:

```
@patient = Patient.find(17)
```

and this in the corresponding view:

```
<%= link_to 'Patient Record', patient_path(@patient) %>
```

then the router will generate the path `/patients/17`. This reduces the brittleness of your view and makes your code easier to understand. Note that the id does not need to be specified in the route helper.

2 Resource Routing: the Rails Default

Resource routing allows you to quickly declare all of the common routes for a given resourceful controller. Instead of declaring separate routes for your `index`, `show`, `new`, `edit`, `create`, `update` and `destroy` actions, a resourceful route declares them in a single line of code.

2.1 Resources on the Web

Browsers request pages from Rails by making a request for a URL using a specific HTTP method, such as `GET`, `POST`, `PATCH`, `PUT` and `DELETE`. Each method is a request to perform an operation on the resource. A resource route maps a number of related requests to actions in a single controller.

When your Rails application receives an incoming request for:

```
DELETE /photos/17
```

it asks the router to map it to a controller action. If the first matching route is:

```
resources :photos
```

Rails would dispatch that request to the `destroy` method on the `photos` controller with `{ id: '17' }` in `params`.

2.2 CRUD, Verbs, and Actions

In Rails, a resourceful route provides a mapping between HTTP verbs and URLs to controller actions. By convention, each action also maps to particular CRUD operations in a database. A single entry in the routing file, such as:

```
resources :photos
```

creates seven different routes in your application, all mapping to the `Photos` controller:

HTTP Verb	Path
GET	/photos
GET	/photos/new
POST	/photos
GET	/photos/:id
GET	/photos/:id/edit
PATCH/PUT	/photos/:id
DELETE	/photos/:id

Note: Because the router uses the HTTP verb and URL to match inbound requests, four URLs map to seven different actions.

Note: Rails routes are matched in the order they are specified, so if you have a `resources :photos` above a `get 'photos/poll'` the `show` action's route for the `resources` line will be matched before the `get` line. To fix this, move the `get` line the `resources` line so that it is matched first.

2.3 Path and URL Helpers

Creating a resourceful route will also expose a number of helpers to the controllers in your application. In the case of `resources :photos`:

- `photos_path` returns `/photos`
- `new_photo_path` returns `/photos/new`
- `edit_photo_path(:id)` returns `/photos/:id/edit` (for instance, `edit_photo_path(10)` returns `/photos/10/edit`)
- `photo_path(:id)` returns `/photos/:id` (for instance, `photo_path(10)` returns `/photos/10`)

Each of these helpers has a corresponding `_url` helper (such as `photos_url`) which returns the same path prefixed with the current host, port and path prefix.

2.4 Defining Multiple Resources at the Same Time

If you need to create routes for more than one resource, you can save a bit of typing by defining them all with a single call to `resources`:

```
resources :photos, :books, :videos
```

This works exactly the same as:

```
resources :photos
resources :books
resources :videos
```

2.5 Singular Resources

Sometimes, you have a resource that clients always look up without referencing an ID. For example, you would like `/profile` to always show the profile of the currently logged in user. In this case, you can use a singular resource to map `/profile` (rather than `/profile/:id`) to the `show` action:

```
get 'profile', to: 'users#show'
```

Passing a `String` to `get` will expect a `controller#action` format, while passing a `Symbol` will map directly to an action:

```
get 'profile', to: :show
```

This resourceful route:

```
resource :geocoder
```

creates six different routes in your application, all mapping to the `Geocoders` controller:

HTTP Verb	Path
GET	/geocoder/new
POST	/geocoder
GET	/geocoder
GET	/geocoder/edit
PATCH/PUT	/geocoder
DELETE	/geocoder

Note: Because you might want to use the same controller for a singular route (`/account`) and a plural route (`/accounts/45`), singular resources map to plural controllers. So that, for example, `resource :photo` and `resources :photos` creates both singular and plural routes that map to the same controller (`PhotosController`).

A singular resourceful route generates these helpers:

- `new_geocoder_path` returns `/geocoder/new`
- `edit_geocoder_path` returns `/geocoder/edit`
- `geocoder_path` returns `/geocoder`

As with plural resources, the same helpers ending in `_url` will also include the host, port and path prefix.

Warning: A long-standing bug prevents `form_for` from working automatically with singular resources. As a workaround, specify the URL for the form directly, like so:

```
form_for @geocoder, url: geocoder_path do |f|
```

2.6 Controller Namespaces and Routing

You may wish to organize groups of controllers under a namespace. Most commonly, you might group a number of administrative controllers under an `Admin::` namespace. You would place these controllers under the `app/controllers/admin` directory, and you can group them together in your router:

```
namespace :admin do
  resources :articles, :comments
end
```

This will create a number of routes for each of the `articles` and `comments` controller. For

`Admin::ArticlesController`, Rails will create:

HTTP Verb	Path
GET	/admin/articles
GET	/admin/articles/new
POST	/admin/articles
GET	/admin/articles/:id
GET	/admin/articles/:id/edit
PATCH/PUT	/admin/articles/:id
DELETE	/admin/articles/:id

If you want to route `/articles` (without the prefix `/admin`) to `Admin::ArticlesController`, you could use:

```
scope module: 'admin' do
  resources :articles, :comments
end
```

or, for a single case:

```
resources :articles, module: 'admin'
```

If you want to route `/admin/articles` to `ArticlesController` (without the `Admin::` module prefix), you could use:

```
scope '/admin' do
  resources :articles, :comments
end
```

or, for a single case:

```
resources :articles, path: '/admin/articles'
```

In each of these cases, the named routes remain the same as if you did not use `scope`. In the last case, the following paths map to `PostsController`:

HTTP Verb	Path
GET	/admin/articles
GET	/admin/articles/new
POST	/admin/articles
GET	/admin/articles/:id
GET	/admin/articles/:id/edit
PATCH/PUT	/admin/articles/:id
DELETE	/admin/articles/:id

Info:

2.7 Nested Resources

It's common to have resources that are logically children of other resources. For example, suppose your application includes these models:

```
class Magazine < ActiveRecord::Base
  has_many :ads
end

class Ad < ActiveRecord::Base
  belongs_to :magazine
end
```

Nested routes allow you to capture this relationship in your routing. In this case, you could include this route declaration:

```
resources :magazines do
  resources :ads
end
```

In addition to the routes for magazines, this declaration will also route ads to an `AdsController`. The ad URLs require a magazine:

HTTP Verb	Path
GET	/magazines/:magazine_id/ads
GET	/magazines/:magazine_id/ads/new
POST	/magazines/:magazine_id/ads
GET	/magazines/:magazine_id/ads/:id
GET	/magazines/:magazine_id/ads/:id/edit
PATCH/PUT	/magazines/:magazine_id/ads/:id
DELETE	/magazines/:magazine_id/ads/:id

This will also create routing helpers such as `magazine_ads_url` and `edit_magazine_ad_path`. These helpers take an instance of Magazine as the first parameter (`magazine_ads_url(@magazine)`).

2.7.1 Limits to Nesting

You can nest resources within other nested resources if you like. For example:

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
```

```
end
```

Deeply-nested resources quickly become cumbersome. In this case, for example, the application would recognize paths such as:

```
/publishers/1/magazines/2/photos/3
```

The corresponding route helper would be `publisher_magazine_photo_url`, requiring you to specify objects at all three levels. Indeed, this situation is confusing enough that a popular article by Jamis Buck proposes a rule of thumb for good Rails design:

Info:

2.7.2 Shallow Nesting

One way to avoid deep nesting (as recommended above) is to generate the collection actions scoped under the parent, so as to get a sense of the hierarchy, but to not nest the member actions. In other words, to only build routes with the minimal amount of information to uniquely identify the resource, like this:

```
resources :articles do
  resources :comments, only: [:index, :new, :create]
end
resources :comments, only: [:show, :edit, :update, :destroy]
```

This idea strikes a balance between descriptive routes and deep nesting. There exists shorthand syntax to achieve just that, via the `:shallow` option:

```
resources :articles do
  resources :comments, shallow: true
end
```

This will generate the exact same routes as the first example. You can also specify the `:shallow` option in the parent resource, in which case all of the nested resources will be shallow:

```
resources :articles, shallow: true do
  resources :comments
  resources :quotes
  resources :drafts
end
```

The `shallow` method of the DSL creates a scope inside of which every nesting is shallow. This generates the same routes as the previous example:

```
shallow do
  resources :articles do
    resources :comments
    resources :quotes
    resources :drafts
  end
```

```
end
```

There exist two options for `scope` to customize shallow routes. `:shallow_path` prefixes member paths with the specified parameter:

```
scope shallow_path: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

The comments resource here will have the following routes generated for it:

HTTP Verb	Path
GET	/articles/:article_id/comments(.:format)
POST	/articles/:article_id/comments(.:format)
GET	/articles/:article_id/comments/new(.:format)
GET	/sekret/comments/:id/edit(.:format)
GET	/sekret/comments/:id(.:format)
PATCH/PUT	/sekret/comments/:id(.:format)
DELETE	/sekret/comments/:id(.:format)

The `:shallow_prefix` option adds the specified parameter to the named helpers:

```
scope shallow_prefix: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

The comments resource here will have the following routes generated for it:

HTTP Verb	Path
GET	/articles/:article_id/comments(.:format)
POST	/articles/:article_id/comments(.:format)
GET	/articles/:article_id/comments/new(.:format)
GET	/comments/:id/edit(.:format)
GET	/comments/:id(.:format)
PATCH/PUT	/comments/:id(.:format)
DELETE	/comments/:id(.:format)

2.8 Routing concerns

Routing Concerns allows you to declare common routes that can be reused inside other resources and

routes. To define a concern:

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end
```

These concerns can be used in resources to avoid code duplication and share behavior across routes:

```
resources :messages, concerns: :commentable

resources :articles, concerns: [:commentable, :image_attachable]
```

The above is equivalent to:

```
resources :messages do
  resources :comments
end

resources :articles do
  resources :comments
  resources :images, only: :index
end
```

Also you can use them in any place that you want inside the routes, for example in a scope or namespace call:

```
namespace :articles do
  concerns :commentable
end
```

2.9 Creating Paths and URLs From Objects

In addition to using the routing helpers, Rails can also create paths and URLs from an array of parameters. For example, suppose you have this set of routes:

```
resources :magazines do
  resources :ads
end
```

When using `magazine_ad_path`, you can pass in instances of `Magazine` and `Ad` instead of the numeric IDs:

```
<%= link_to 'Ad details', magazine_ad_path(@magazine, @ad) %>
```

You can also use `url_for` with a set of objects, and Rails will automatically determine which route you want:

```
<%= link_to 'Ad details', url_for([@magazine, @ad]) %>
```

In this case, Rails will see that `@magazine` is a `Magazine` and `@ad` is an `Ad` and will therefore use the `magazine_ad_path` helper. In helpers like `link_to`, you can specify just the object in place of the full `url_for` call:

```
<%= link_to 'Ad details', [@magazine, @ad] %>
```

If you wanted to link to just a magazine:

```
<%= link_to 'Magazine details', @magazine %>
```

For other actions, you just need to insert the action name as the first element of the array:

```
<%= link_to 'Edit Ad', [:edit, @magazine, @ad] %>
```

This allows you to treat instances of your models as URLs, and is a key advantage to using the resourceful style.

2.10 Adding More RESTful Actions

You are not limited to the seven routes that RESTful routing creates by default. If you like, you may add additional routes that apply to the collection or individual members of the collection.

2.10.1 Adding Member Routes

To add a member route, just add a `member` block into the resource block:

```
resources :photos do
  member do
    get 'preview'
  end
end
```

This will recognize `/photos/1/preview` with GET, and route to the `preview` action of `PhotosController`, with the resource id value passed in `params[:id]`. It will also create the `preview_photo_url` and `preview_photo_path` helpers.

Within the block of member routes, each route name specifies the HTTP verb will be recognized. You can use `get`, `patch`, `put`, `post`, or `delete` here . If you don't have multiple `member` routes, you can also pass `:on` to a route, eliminating the block:

```
resources :photos do
  get 'preview', on: :member
end
```

You can leave out the `:on` option, this will create the same member route except that the resource id value will be available in `params[:photo_id]` instead of `params[:id]`.

2.10.2 Adding Collection Routes

To add a route to the collection:

```
resources :photos do
  collection do
    get 'search'
  end
end
```

This will enable Rails to recognize paths such as `/photos/search` with GET, and route to the `search` action of `PhotosController`. It will also create the `search_photos_url` and `search_photos_path` route helpers.

Just as with member routes, you can pass `:on` to a route:

```
resources :photos do
  get 'search', on: :collection
end
```

2.10.3 Adding Routes for Additional New Actions

To add an alternate new action using the `:on` shortcut:

```
resources :comments do
  get 'preview', on: :new
end
```

This will enable Rails to recognize paths such as `/comments/new/preview` with GET, and route to the `preview` action of `CommentsController`. It will also create the `preview_new_comment_url` and `preview_new_comment_path` route helpers.

Info: If you find yourself adding many extra actions to a resourceful route, it's time to stop and ask yourself whether you're disguising the presence of another resource.

3 Non-Resourceful Routes

In addition to resource routing, Rails has powerful support for routing arbitrary URLs to actions. Here, you don't get groups of routes automatically generated by resourceful routing. Instead, you set up each route within your application separately.

While you should usually use resourceful routing, there are still many places where the simpler routing is more appropriate. There's no need to try to shoehorn every last piece of your application into a resourceful framework if that's not a good fit.

In particular, simple routing makes it very easy to map legacy URLs to new Rails actions.

3.1 Bound Parameters

When you set up a regular route, you supply a series of symbols that Rails maps to parts of an incoming HTTP request. Two of these symbols are special: `:controller` maps to the name of a controller in your application, and `:action` maps to the name of an action within that controller. For example, consider this route:

```
get ':controller(/:action(/:id))'
```

If an incoming request of `/photos/show/1` is processed by this route (because it hasn't matched any previous route in the file), then the result will be to invoke the `show` action of the `PhotosController`, and to make the final parameter `"1"` available as `params[:id]`. This route will also route the incoming request of `/photos` to `PhotosController#index`, since `:action` and `:id` are optional parameters, denoted by parentheses.

3.2 Dynamic Segments

You can set up as many dynamic segments within a regular route as you like. Anything other than `:controller` or `:action` will be available to the action as part of `params`. If you set up this route:

```
get ':controller/:action/:id/:user_id'
```

An incoming path of `/photos/show/1/2` will be dispatched to the `show` action of the `PhotosController`. `params[:id]` will be `"1"`, and `params[:user_id]` will be `"2"`.

Note: You can't use `:namespace` or `:module` with a `:controller` path segment. If you need to do this then use a constraint on :controller that matches the namespace you require. e.g:

```
get ':controller(/:action(/:id))', controller: /admin\/[^\/]+/
```

Info: By default, dynamic segments don't accept dots - this is because the dot is used as a separator

for formatted routes. If you need to use a dot within a dynamic segment, add a constraint that overrides this – for example, `id: /[^\/]+/` allows anything except a slash.

3.3 Static Segments

You can specify static segments when creating a route by not prepending a colon to a fragment:

```
get ':controller/:action/:id/with_user/:user_id'
```

This route would respond to paths such as `/photos/show/1/with_user/2`. In this case, `params` would be `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`.

3.4 The Query String

The `params` will also include any parameters from the query string. For example, with this route:

```
get ':controller/:action/:id'
```

An incoming path of `/photos/show/1?user_id=2` will be dispatched to the `show` action of the `Photos` controller. `params` will be `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`.

3.5 Defining Defaults

You do not need to explicitly use the `:controller` and `:action` symbols within a route. You can supply them as defaults:

```
get 'photos/:id', to: 'photos#show'
```

With this route, Rails will match an incoming path of `/photos/12` to the `show` action of `PhotosController`.

You can also define other defaults in a route by supplying a hash for the `:defaults` option. This even applies to parameters that you do not specify as dynamic segments. For example:

```
get 'photos/:id', to: 'photos#show', defaults: { format: 'jpg' }
```

Rails would match `photos/12` to the `show` action of `PhotosController`, and set `params[:format]` to `"jpg"`.

3.6 Naming Routes

You can specify a name for any route using the `:as` option:

```
get 'exit', to: 'sessions#destroy', as: :logout
```

This will create `logout_path` and `logout_url` as named helpers in your application. Calling `logout_path` will return `/exit`

You can also use this to override routing methods defined by resources, like this:

```
get ':username', to: 'users#show', as: :user
```

This will define a `user_path` method that will be available in controllers, helpers and views that will go to a route such as `/bob`. Inside the `show` action of `UsersController`, `params[:username]` will contain the username for the user. Change `:username` in the route definition if you do not want your parameter name to be `:username`.

3.7 HTTP Verb Constraints

In general, you should use the `get`, `post`, `put`, `patch` and `delete` methods to constrain a route to a particular verb. You can use the `match` method with the `:via` option to match multiple verbs at once:

```
match 'photos', to: 'photos#show', via: [:get, :post]
```

You can match all verbs to a particular route using `via: :all`:

```
match 'photos', to: 'photos#show', via: :all
```

Note: Routing both `GET` and `POST` requests to a single action has security implications. In general, you should avoid routing all verbs to an action unless you have a good reason to.

Note: 'GET' in Rails won't check for CSRF token. You should never write to the database from 'GET' requests, for more information see the security guide on CSRF countermeasures.

3.8 Segment Constraints

You can use the `:constraints` option to enforce a format for a dynamic segment:

```
get 'photos/:id', to: 'photos#show', constraints: { id: /[A-Z]\d{5}/ }
```

This route would match paths such as `/photos/A12345`, but not `/photos/893`. You can more succinctly express the same route this way:

```
get 'photos/:id', to: 'photos#show', id: /[A-Z]\d{5}/
```

`:constraints` takes regular expressions with the restriction that regexp anchors can't be used. For example, the following route will not work:

```
get '/:id', to: 'articles#show', constraints: { id: /^\d/ }
```

However, note that you don't need to use anchors because all routes are anchored at the start.

For example, the following routes would allow for `articles` with `to_param` values like `1-hello-world` that always begin with a number and `users` with `to_param` values like `david` that never begin with a number to share the root namespace:

```
get '/:id', to: 'articles#show', constraints: { id: /\d.+/ }
get '/:username', to: 'users#show'
```

3.9 Request-Based Constraints

You can also constrain a route based on any method on the Request object that returns a `String`.

You specify a request-based constraint the same way that you specify a segment constraint:

```
get 'photos', to: 'photos#index', constraints: { subdomain: 'admin' }
```

You can also specify constraints in a block form:

```
namespace :admin do
  constraints subdomain: 'admin' do
    resources :photos
  end
end
```

Note: Request constraints work by calling a method on the Request object with the same name as the hash key and then compare the return value with the hash value. Therefore, constraint values should match the corresponding Request object method return type. For example: `constraints: { subdomain: 'api' }` will match an `api` subdomain as expected, however using a symbol `constraints: { subdomain: :api }` will not, because `request.subdomain` returns `'api'` as a String.

3.10 Advanced Constraints

If you have a more advanced constraint, you can provide an object that responds to `matches?` that Rails should use. Let's say you wanted to route all users on a blacklist to the `BlacklistController`. You could do:

```
class BlacklistConstraint
  def initialize
    @ips = Blacklist.retrieve_ips
  end

  def matches?(request)
    @ips.include?(request.remote_ip)
  end
end

Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: BlacklistConstraint.new
```

```
end
```

You can also specify constraints as a lambda:

```
Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: lambda { |request| Blacklist.retrieve_ips.include?(request.remot
end
```

Both the `matches?` method and the lambda gets the `request` object as an argument.

3.11 Route Globbing and Wildcard Segments

Route globbing is a way to specify that a particular parameter should be matched to all the remaining parts of a route. For example:

```
get 'photos/*other', to: 'photos#unknown'
```

This route would match `photos/12` or `/photos/long/path/to/12`, setting `params[:other]` to `"12"` or `"long/path/to/12"`. The fragments prefixed with a star are called "wildcard segments".

Wildcard segments can occur anywhere in a route. For example:

```
get 'books/*section/:title', to: 'books#show'
```

would match `books/some/section/last-words-a-memoir` with `params[:section]` equals `'some/section'`, and `params[:title]` equals `'last-words-a-memoir'`.

Technically, a route can have even more than one wildcard segment. The matcher assigns segments to parameters in an intuitive way. For example:

```
get '*a/foo/*b', to: 'test#index'
```

would match `zoo/woo/foo/bar/baz` with `params[:a]` equals `'zoo/woo'`, and `params[:b]` equals `'bar/baz'`.

Note: By requesting `'/foo/bar.json'`, your `params[:pages]` will be equal to `'foo/bar'` with the request format of JSON. If you want the old 3.0.x behavior back, you could supply `format: false` like this:

```
get '*pages', to: 'pages#show', format: false
```

Note: If you want to make the format segment mandatory, so it cannot be omitted, you can supply `format: true` like this:

```
get '*pages', to: 'pages#show', format: true
```

3.12 Redirection

You can redirect any path to another path using the `redirect` helper in your router:

```
get '/stories', to: redirect('/articles')
```

You can also reuse dynamic segments from the match in the path to redirect to:

```
get '/stories/:name', to: redirect('/articles/%{name}')
```

You can also provide a block to redirect, which receives the symbolized path parameters and the request object:

```
get '/stories/:name', to: redirect { |path_params, req| "/articles/#{path_params
get '/stories', to: redirect { |path_params, req| "/articles/#{req.subdomain}" }
```

Please note that this redirection is a 301 "Moved Permanently" redirect. Keep in mind that some web browsers or proxy servers will cache this type of redirect, making the old page inaccessible.

In all of these cases, if you don't provide the leading host (`http://www.example.com`), Rails will take those details from the current request.

3.13 Routing to Rack Applications

Instead of a String like `'articles#index'`, which corresponds to the `index` action in the `ArticlesController`, you can specify any Rack application as the endpoint for a matcher:

```
match '/application.js', to: Sprockets, via: :all
```

As long as `Sprockets` responds to `call` and returns a `[status, headers, body]`, the router won't know the difference between the Rack application and an action. This is an appropriate use of `via: :all`, as you will want to allow your Rack application to handle all verbs as it considers appropriate.

Note: For the curious, `'articles#index'` actually expands out to `ArticlesController.action(:index)`, which returns a valid Rack application.

3.14 Using `root`

You can specify what Rails should route `'/'` to with the `root` method:

```
root to: 'pages#main'
root 'pages#main' # shortcut for the above
```

You should put the `root` route at the top of the file, because it is the most popular route and should be matched first.

Note: The `root` route only routes `GET` requests to the action.

You can also use root inside namespaces and scopes as well. For example:

```
namespace :admin do
  root to: "admin#index"
end

root to: "home#index"
```

3.15 Unicode character routes

You can specify unicode character routes directly. For example:

```
get 'こんにちは', to: 'welcome#index'
```

4 Customizing Resourceful Routes

While the default routes and helpers generated by `resources :articles` will usually serve you well, you may want to customize them in some way. Rails allows you to customize virtually any generic part of the resourceful helpers.

4.1 Specifying a Controller to Use

The `:controller` option lets you explicitly specify a controller to use for the resource. For example:

```
resources :photos, controller: 'images'
```

will recognize incoming paths beginning with `/photos` but route to the `Images` controller:

HTTP Verb	Path
GET	/photos
GET	/photos/new
POST	/photos
GET	/photos/:id
GET	/photos/:id/edit
PATCH/PUT	/photos/:id
DELETE	/photos/:id

Note: Use `photos_path`, `new_photo_path`, etc. to generate paths for this resource.

For namespaced controllers you can use the directory notation. For example:

```
resources :user_permissions, controller: 'admin/user_permissions'
```

This will route to the `Admin::UserPermissions` controller.

Note: Only the directory notation is supported. Specifying the controller with Ruby constant notation (eg. `controller: 'Admin::UserPermissions'`) can lead to routing problems and results in a warning.

4.2 Specifying Constraints

You can use the `:constraints` option to specify a required format on the implicit `id`. For example:

```
resources :photos, constraints: { id: /[A-Z][A-Z][0-9]+/ }
```

This declaration constrains the `:id` parameter to match the supplied regular expression. So, in this

case, the router would no longer match `/photos/1` to this route. Instead, `/photos/RR27` would match.

You can specify a single constraint to apply to a number of routes by using the block form:

```
constraints(id: /[A-Z][A-Z][0-9]+/) do
  resources :photos
  resources :accounts
end
```

Note: Of course, you can use the more advanced constraints available in non-resourceful routes in this context.

Info: By default the `:id` parameter doesn't accept dots - this is because the dot is used as a separator for formatted routes. If you need to use a dot within an `:id` add a constraint which overrides this - for example `id: /[^\/]+/` allows anything except a slash.

4.3 Overriding the Named Helpers

The `:as` option lets you override the normal naming for the named route helpers. For example:

```
resources :photos, as: 'images'
```

will recognize incoming paths beginning with `/photos` and route the requests to `PhotosController`, but use the value of the :as option to name the helpers.

HTTP Verb	Path
GET	/photos
GET	/photos/new
POST	/photos
GET	/photos/:id
GET	/photos/:id/edit
PATCH/PUT	/photos/:id
DELETE	/photos/:id

4.4 Overriding the `new` and `edit` Segments

The `:path_names` option lets you override the automatically-generated `new` and `edit` segments in paths:

```
resources :photos, path_names: { new: 'make', edit: 'change' }
```

This would cause the routing to recognize paths such as:

```
/photos/make
```

```
/photos/1/change
```

Note: The actual action names aren't changed by this option. The two paths shown would still route to the `new` and `edit` actions.

Info: If you find yourself wanting to change this option uniformly for all of your routes, you can use a scope.

```
scope path_names: { new: 'make' } do
  # rest of your routes
end
```

4.5 Prefixing the Named Route Helpers

You can use the `:as` option to prefix the named route helpers that Rails generates for a route. Use this option to prevent name collisions between routes using a path scope. For example:

```
scope 'admin' do
  resources :photos, as: 'admin_photos'
end

resources :photos
```

This will provide route helpers such as `admin_photos_path`, `new_admin_photo_path`, etc.

To prefix a group of route helpers, use `:as` with `scope`:

```
scope 'admin', as: 'admin' do
  resources :photos, :accounts
end

resources :photos, :accounts
```

This will generate routes such as `admin_photos_path` and `admin_accounts_path` which map to `/admin/photos` and `/admin/accounts` respectively.

Note: The `namespace` scope will automatically add `:as` as well as `:module` and `:path` prefixes.

You can prefix routes with a named parameter also:

```
scope ':username' do
  resources :articles
end
```

This will provide you with URLs such as `/bob/articles/1` and will allow you to reference the `username` part of the path as `params[:username]` in controllers, helpers and views.

4.6 Restricting the Routes Created

By default, Rails creates routes for the seven default actions (`index`, `show`, `new`, `create`, `edit`, `update`, and `destroy`) for every RESTful route in your application. You can use the `:only` and `:except` options to fine-tune this behavior. The `:only` option tells Rails to create only the specified routes:

```
resources :photos, only: [:index, :show]
```

Now, a `GET` request to `/photos` would succeed, but a `POST` request to `/photos` (which would ordinarily be routed to the `create` action) will fail.

The `:except` option specifies a route or list of routes that Rails should create:

```
resources :photos, except: :destroy
```

In this case, Rails will create all of the normal routes except the route for `destroy` (a `DELETE` request to `/photos/:id`).

Info: If your application has many RESTful routes, using `:only` and `:except` to generate only the routes that you actually need can cut down on memory use and speed up the routing process.

4.7 Translated Paths

Using `scope`, we can alter path names generated by resources:

```
scope(path_names: { new: 'neu', edit: 'bearbeiten' }) do
  resources :categories, path: 'kategorien'
end
```

Rails now creates routes to the `CategoriesController`.

HTTP Verb	Path
GET	/kategorien
GET	/kategorien/neu
POST	/kategorien
GET	/kategorien/:id
GET	/kategorien/:id/bearbeiten
PATCH/PUT	/kategorien/:id
DELETE	/kategorien/:id

4.8 Overriding the Singular Form

If you want to define the singular form of a resource, you should add additional rules to the `Inflector`:

```
ActiveSupport::Inflector.inflections do |inflect|
```

```
  inflect.irregular 'tooth', 'teeth'
end
```

4.9 Using `:as` in Nested Resources

The `:as` option overrides the automatically-generated name for the resource in nested route helpers. For example:

```
resources :magazines do
  resources :ads, as: 'periodical_ads'
end
```

This will create routing helpers such as `magazine_periodical_ads_url` and `edit_magazine_periodical_ad_path`.

4.10 Overriding Named Route Parameters

The `:param` option overrides the default resource identifier `:id` (name of the dynamic segment used to generate the routes). You can access that segment from your controller using `params[<:param>]`.

```
resources :videos, param: :identifier
```

```
videos GET  /videos(.:format)                  videos#index
       POST /videos(.:format)                       videos#create
 new_videos GET  /videos/new(.:format)                  videos#new
edit_videos GET  /videos/:identifier/edit(.:format) videos#edit
```

```
Video.find_by(identifier: params[:identifier])
```

5 Inspecting and Testing Routes

Rails offers facilities for inspecting and testing your routes.

5.1 Listing Existing Routes

To get a complete list of the available routes in your application, visit `http://localhost:3000/rails/info/routes` in your browser while your server is running in the environment. You can also execute the `rake routes` command in your terminal to produce the same output.

Both methods will list all of your routes, in the same order that they appear in `routes.rb`. For each route, you'll see:

- The route name (if any)
- The HTTP verb used (if the route doesn't respond to all verbs)
- The URL pattern to match
- The routing parameters for the route

For example, here's a small section of the `rake routes` output for a RESTful route:

```
users GET    /users(.:format)          users#index
      POST   /users(.:format)          users#create
 new_user GET    /users/new(.:format)      users#new
edit_user GET    /users/:id/edit(.:format) users#edit
```

You may restrict the listing to the routes that map to a particular controller setting the `CONTROLLER` environment variable:

```
$ CONTROLLER=users bin/rake routes
```

Info: You'll find that the output from `rake routes` is much more readable if you widen your terminal window until the output lines don't wrap.

5.2 Testing Routes

Routes should be included in your testing strategy (just like the rest of your application). Rails offers three built-in assertions designed to make testing routes simpler:

- `assert_generates`
- `assert_recognizes`
- `assert_routing`

5.2.1 The `assert_generates` Assertion

`assert_generates` asserts that a particular set of options generate a particular path and can be used with default routes or custom routes. For example:

```
assert_generates '/photos/1', { controller: 'photos', action: 'show', id: '1' }
assert_generates '/about', controller: 'pages', action: 'about'
```

5.2.2 The `assert_recognizes` Assertion

`assert_recognizes` is the inverse of `assert_generates`. It asserts that a given path is recognized and routes it to a particular spot in your application. For example:

```
assert_recognizes({ controller: 'photos', action: 'show', id: '1' }, '/photos/1'
```

You can supply a `:method` argument to specify the HTTP verb:

```
assert_recognizes({ controller: 'photos', action: 'create' }, { path: 'photos', r
```

5.2.3 The `assert_routing` Assertion

The `assert_routing` assertion checks the route both ways: it tests that the path generates the options, and that the options generate the path. Thus, it combines the functions of `assert_generates` and `assert_recognizes`:

```
assert_routing({ path: 'photos', method: :post }, { controller: 'photos', action
```

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.