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 `config/routes.rb`.
- How to construct your own routes, using either the preferred resourceful style or the `match` method.
- How to declare route parameters, which are passed onto controller actions.
- How to automatically create paths and URLs using route helpers.
- Advanced techniques such as creating constraints and mounting Rack endpoints.

The Purpose of the Rails Router

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

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:

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

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

NOTE: Rails uses snake_case for controller names here, if you have a multiple word controller like `MonsterTrucksController`, you want to use `monster_trucks#show` for example.

Generating Paths and URLs from Code

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

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

and your application contains this code in the controller:

```ruby
@patient = Patient.find(params[:id])
```

and this in the corresponding view:

```erb
<%= 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.

Configuring the Rails Router

The routes for your application or engine live in the file `config/routes.rb` and typically looks like this:

```ruby
Rails.application.routes.draw do
  resources :brands, only: [:index, :show] do
    resources :products, only: [:index, :show]
  end

  resource :basket, only: [:show, :update, :destroy]

  resolve("Basket") { route_for(:basket) }
end
```

Since this is a regular Ruby source file you can use all of its features to help you define your routes but be careful with variable names as they can clash with the DSL methods of the router.

NOTE: The `Rails.application.routes.draw do ... end` block that wraps your route definitions is required to establish the scope for the router DSL and must not be deleted.

Resource Routing: the Rails Default

Resource routing allows you to quickly declare all of the common routes for a given resourceful controller. A single call to `resources` can declare all of the necessary routes for your `index`, `show`, `new`, `edit`, `create`, `update`, and `destroy` actions.

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:

```ruby
resources :photos
```

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

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 a specific CRUD operation 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	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

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 **above** the `resources` line so that it is matched first.

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.

TIP: To find the route helper names for your routes, see Listing existing routes below.

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
```

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 `to:` will expect a `controller#action` format. When using a `Symbol` , the `to:` option should be replaced with `action:` . When using a `String` without a `#` , the `to:` option should be replaced with `controller:` :

```
get 'profile', action: :show, controller: 'users'
```

This resourceful route:

```
resource :geocoder
resolve('Geocoder') { [:geocoder] }
```

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

HTTP Verb	Path	Controller#Action	Used for
GET	/geocoder/new	geocoders#new	return an HTML form for creating the geocoder
POST	/geocoder	geocoders#create	create the new geocoder
GET	/geocoder	geocoders#show	display the one and only geocoder resource
GET	/geocoder/edit	geocoders#edit	return an HTML form for editing the geocoder
PATCH/PUT	/geocoder	geocoders#update	update the one and only geocoder resource
DELETE	/geocoder	geocoders#destroy	delete the geocoder resource

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`

NOTE: The call to `resolve` is necessary for converting instances of the `Geocoder` to routes through record identification.

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

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, and place these controllers under the `app/controllers/admin` directory. You can route to such a group by using a `namespace` block:

```
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	Controller#Action	Named Route Helper
GET	/admin/articles	admin/articles#index	admin_articles_path
GET	/admin/articles/new	admin/articles#new	new_admin_article_path
POST	/admin/articles	admin/articles#create	admin_articles_path
GET	/admin/articles/:id	admin/articles#show	admin_article_path(:id)
GET	/admin/articles/:id/edit	admin/articles#edit	edit_admin_article_path(:id)
PATCH/PUT	/admin/articles/:id	admin/articles#update	admin_article_path(:id)
DELETE	/admin/articles/:id	admin/articles#destroy	admin_article_path(:id)

If instead you want to route `/articles` (without the prefix `/admin`) to `Admin::ArticlesController`, you can specify the module with a `scope` block:

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

This can also be done for a single route:

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

If instead you want to route `/admin/articles` to `ArticlesController` (without the `Admin::` module prefix), you can specify the path with a `scope` block:

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

This can also be done for a single route:

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

In both of these cases, the named route helpers remain the same as if you did not use `scope`. In the last case, the following paths map to `ArticlesController`:

HTTP Verb	Path	Controller#Action	Named Route Helper
GET	/admin/articles	articles#index	articles_path
GET	/admin/articles/new	articles#new	new_article_path
POST	/admin/articles	articles#create	articles_path
GET	/admin/articles/:id	articles#show	article_path(:id)
GET	/admin/articles/:id/edit	articles#edit	edit_article_path(:id)
PATCH/PUT	/admin/articles/:id	articles#update	article_path(:id)
DELETE	/admin/articles/:id	articles#destroy	article_path(:id)

TIP: If you need to use a different controller namespace inside a `namespace` block you can specify an absolute controller path, e.g: `get '/foo', to: '/foo#index'`.

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 < ApplicationRecord
  has_many :ads
end

class Ad < ApplicationRecord
  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	Controller#Action	Used for
GET	/magazines/:magazine_id/ads	ads#index	display a list of all ads for a

			specific magazine
GET	/magazines/:magazine_id/ads/new	ads#new	return an HTML form for creating a new ad belonging to a specific magazine
POST	/magazines/:magazine_id/ads	ads#create	create a new ad belonging to a specific magazine
GET	/magazines/:magazine_id/ads/:id	ads#show	display a specific ad belonging to a specific magazine
GET	/magazines/:magazine_id/ads/:id/edit	ads#edit	return an HTML form for editing an ad belonging to a specific magazine
PATCH/PUT	/magazines/:magazine_id/ads/:id	ads#update	update a specific ad belonging to a specific magazine
DELETE	/magazines/:magazine_id/ads/:id	ads#destroy	delete a specific ad belonging to a specific magazine

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)`).

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:

TIP: Resources should never be nested more than 1 level deep.

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 articles resource here will have the following routes generated for it:

HTTP Verb	Path	Controller#Action	Named Route Helper
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_pat
GET	/comments/:id/edit(.:format)	comments#edit	edit_comment_path
GET	/comments/:id(.:format)	comments#show	comment_path
PATCH/PUT	/comments/:id(.:format)	comments#update	comment_path
DELETE	/comments/:id(.:format)	comments#destroy	comment_path
GET	/articles/:article_id/quotes(.:format)	quotes#index	article_quotes_path
POST	/articles/:article_id/quotes(.:format)	quotes#create	article_quotes_path
GET	/articles/:article_id/quotes/new(.:format)	quotes#new	new_article_quote_path
GET	/quotes/:id/edit(.:format)	quotes#edit	edit_quote_path
GET	/quotes/:id(.:format)	quotes#show	quote_path
PATCH/PUT	/quotes/:id(.:format)	quotes#update	quote_path
DELETE	/quotes/:id(.:format)	quotes#destroy	quote_path
GET	/articles/:article_id/drafts(.:format)	drafts#index	article_drafts_path

POST	/articles/:article_id/drafts(.:format)	drafts#create	article_drafts_path
GET	/articles/:article_id/drafts/new(.:format)	drafts#new	new_article_draft_path
GET	/drafts/:id/edit(.:format)	drafts#edit	edit_draft_path
GET	/drafts/:id(.:format)	drafts#show	draft_path
PATCH/PUT	/drafts/:id(.:format)	drafts#update	draft_path
DELETE	/drafts/:id(.:format)	drafts#destroy	draft_path
GET	/articles(.:format)	articles#index	articles_path
POST	/articles(.:format)	articles#create	articles_path
GET	/articles/new(.:format)	articles#new	new_article_path
GET	/articles/:id/edit(.:format)	articles#edit	edit_article_path
GET	/articles/:id(.:format)	articles#show	article_path
PATCH/PUT	/articles/:id(.:format)	articles#update	article_path
DELETE	/articles/:id(.:format)	articles#destroy	article_path

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	Controller#Action	Named Route Helper
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_pat

GET	/sekret/comments/:id/edit(.:format)	comments#edit	edit_comment_path
GET	/sekret/comments/:id(.:format)	comments#show	comment_path
PATCH/PUT	/sekret/comments/:id(.:format)	comments#update	comment_path
DELETE	/sekret/comments/:id(.:format)	comments#destroy	comment_path

The `:shallow_prefix` option adds the specified parameter to the named route 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	Controller#Action	Named Route Helper
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_pat
GET	/comments/:id/edit(.:format)	comments#edit	edit_sekret_comment_pat
GET	/comments/:id(.:format)	comments#show	sekret_comment_path
PATCH/PUT	/comments/:id(.:format)	comments#update	sekret_comment_path
DELETE	/comments/:id(.:format)	comments#destroy	sekret_comment_path

Routing Concerns

Routing concerns allow you to declare common routes that can be reused inside other resources and routes. To define a concern, use a `concern` block:

```
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
```

You can also use them anywhere by calling `concerns`. For example, in a `scope` or `namespace` block:

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

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.

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.

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 that 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]`. Route helpers will also be renamed from `preview_photo_url` and `preview_photo_path` to `photo_preview_url` and `photo_preview_path`.

Adding Collection Routes

To add a route to the collection, use a `collection` block:

```
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
```

NOTE: If you're defining additional resource routes with a symbol as the first positional argument, be mindful that it is not equivalent to using a string. Symbols infer controller actions while strings infer paths.

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.

TIP: 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.

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 separately within your application.

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.

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. For example, consider this route:

```
get 'photos(/:id)', to: 'photos#display'
```

If an incoming request of `/photos/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 `display` 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#display`, since `:id` is an optional parameter, denoted by parentheses.

Dynamic Segments

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

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

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

TIP: 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.

Static Segments

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

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

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

The Query String

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

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

An incoming path of `/photos/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' }`.

Defining Defaults

You can define 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"`.

You can also use a `defaults` block to define the defaults for multiple items:

```
defaults format: :json do
  resources :photos
end
```

NOTE: You cannot override defaults via query parameters - this is for security reasons. The only defaults that can be overridden are dynamic segments via substitution in the URL path.

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 route helpers in your application. Calling `logout_path` will return `/exit`

You can also use this to override routing methods defined by resources by placing custom routes before the resource is defined, like this:

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

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`.

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.

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 and the end.

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'
```

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 by using a `constraints` block:

```
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 comparing 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.

NOTE: There is an exception for the `format` constraint: while it's a method on the Request object, it's also an implicit optional parameter on every path. Segment constraints take precedence and the `format` constraint is only applied as such when enforced through a hash. For example, `get 'foo', constraints: { format: 'json' }` will match `GET /foo` because the format is optional by default. However, you can use a lambda like in `get 'foo', constraints: lambda { |req| req.format == :json }` and the route will only match explicit JSON requests.

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 restricted list to the `RestrictedListController`. You could do:

```
class RestrictedListConstraint
  def initialize
    @ips = RestrictedList.retrieve_ips
  end

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

```
end

Rails.application.routes.draw do
  get '*path', to: 'restricted_list#index',
    constraints: RestrictedListConstraint.new
end
```

You can also specify constraints as a lambda:

```
Rails.application.routes.draw do
  get '*path', to: 'restricted_list#index',
    constraints: lambda { |request| RestrictedList.retrieve_ips.include?
(request.remote_ip) }
end
```

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

Constraints in a block form

You can specify constraints in a block form. This is useful for when you need to apply the same rule to several routes. For example:

```
class RestrictedListConstraint
  # ...Same as the example above
end

Rails.application.routes.draw do
  constraints(RestrictedListConstraint.new) do
    get '*path', to: 'restricted_list#index'
    get '*other-path', to: 'other_restricted_list#index'
  end
end
```

You can also use a `lambda`:

```
Rails.application.routes.draw do
  constraints(lambda { |request| RestrictedList.retrieve_ips.include?
(request.remote_ip) }) do
    get '*path', to: 'restricted_list#index'
    get '*other-path', to: 'other_restricted_list#index'
  end
end
```

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 segments 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
```

Redirection

You can redirect any path to another path by 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[:name].pluralize}" }
get '/stories', to: redirect { |path_params, req| "/articles/#{req.subdomain}" }
```

Please note that default 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. You can use the `:status` option to change the response status:

```
get '/stories/:name', to: redirect('/articles/%{name}', status: 302)
```

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.

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: MyRackApp, via: :all
```

As long as `MyRackApp` 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.

NOTE: Since procs/lambdas are objects that respond to `call` , you can implement very simple routes (e.g. for health checks) inline:
`get '/health', to: ->(env) { [204, {}, ['']] }`

If you specify a Rack application as the endpoint for a matcher, remember that the route will be unchanged in the receiving application. With the following route your Rack application should expect the route to be `/admin` :

```
match '/admin', to: AdminApp, via: :all
```

If you would prefer to have your Rack application receive requests at the root path instead, use `mount` :

```
mount AdminApp, at: '/admin'
```

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"
```

Unicode Character Routes

You can specify unicode character routes directly. For example:

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

Direct Routes

You can create custom URL helpers directly by calling `direct`. For example:

```
direct :homepage do
  "https://rubyonrails.org"
end

# >> homepage_url
# => "https://rubyonrails.org"
```

The return value of the block must be a valid argument for the `url_for` method. So, you can pass a valid string URL, Hash, Array, an Active Model instance, or an Active Model class.

```
direct :commentable do |model|
  [ model, anchor: model.dom_id ]
end

direct :main do
  { controller: 'pages', action: 'index', subdomain: 'www' }
end
```

Using `resolve`

The `resolve` method allows customizing polymorphic mapping of models. For example:

```
resource :basket

resolve("Basket") { [:basket] }
```

```
<%= form_with model: @basket do |form| %>
  <!-- basket form -->
<% end %>
```

This will generate the singular URL `/basket` instead of the usual `/baskets/:id`.

Customizing Resourceful Routes

While the default routes and helpers generated by `resources` 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.

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	Controller#Action	Named Route Helper
GET	/photos	images#index	photos_path
GET	/photos/new	images#new	new_photo_path
POST	/photos	images#create	photos_path
GET	/photos/:id	images#show	photo_path(:id)
GET	/photos/:id/edit	images#edit	edit_photo_path(:id)
PATCH/PUT	/photos/:id	images#update	photo_path(:id)
DELETE	/photos/:id	images#destroy	photo_path(: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 (e.g. `controller: 'Admin::UserPermissions'`) can lead to routing problems and results in a warning.

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.

TIP: 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.

Overriding the Named Route 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	Controller#Action	Named Route Helper
GET	/photos	photos#index	images_path
GET	/photos/new	photos#new	new_image_path
POST	/photos	photos#create	images_path
GET	/photos/:id	photos#show	image_path(:id)
GET	/photos/:id/edit	photos#edit	edit_image_path(:id)
PATCH/PUT	/photos/:id	photos#update	image_path(:id)
DELETE	/photos/:id	photos#destroy	image_path(:id)

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.

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

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

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.

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 *not* 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`).

TIP: 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.

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	Controller#Action	Named Route Helper
GET	/kategorien	categories#index	categories_path
GET	/kategorien/neu	categories#new	new_category_path
POST	/kategorien	categories#create	categories_path
GET	/kategorien/:id	categories#show	category_path(:id)
GET	/kategorien/:id/bearbeiten	categories#edit	edit_category_path(:id)
PATCH/PUT	/kategorien/:id	categories#update	category_path(:id)
DELETE	/kategorien/:id	categories#destroy	category_path(:id)

Overriding the Singular Form

If you want to override the singular form of a resource, you should add additional rules to the inflector via `inflections`:

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tooth', 'teeth'
end
```

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`.

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_video GET  /videos/new(.:format)              videos#new
edit_video GET  /videos/:identifier/edit(.:format) videos#edit
```

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

You can override `ActiveRecord::Base#to_param` of the associated model to construct a URL:

```
class Video < ApplicationRecord
  def to_param
    identifier
  end
end
```

```
video = Video.find_by(identifier: "Roman-Holiday")
edit_video_path(video) # => "/videos/Roman-Holiday/edit"
```

Breaking up *very* large route file into multiple small ones:

If you work in a large application with thousands of routes, a single `config/routes.rb` file can become cumbersome and hard to read.

Rails offers a way to break a gigantic single `routes.rb` file into multiple small ones using the `draw` macro.

You could have an `admin.rb` route that contains all the routes for the admin area, another `api.rb` file for API related resources, etc.

```
# config/routes.rb

Rails.application.routes.draw do
  get 'foo', to: 'foo#bar'

  draw(:admin) # Will load another route file located in `config/routes/admin.rb`
end
```

```
# config/routes/admin.rb

namespace :admin do
```

```
  resources :comments
end
```

Calling `draw(:admin)` inside the `Rails.application.routes.draw` block itself will try to load a route file that has the same name as the argument given (`admin.rb` in this example). The file needs to be located inside the `config/routes` directory or any sub-directory (i.e. `config/routes/admin.rb` or `config/routes/external/admin.rb`).

You can use the normal routing DSL inside the `admin.rb` routing file, but you **shouldn't** surround it with the `Rails.application.routes.draw` block like you did in the main `config/routes.rb` file.

Don't use this feature unless you really need it

Having multiple routing files makes discoverability and understandability harder. For most applications - even those with a few hundred routes - it's easier for developers to have a single routing file. The Rails routing DSL already offers a way to break routes in an organized manner with `namespace` and `scope` .

Inspecting and Testing Routes

Rails offers facilities for inspecting and testing your routes.

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 **development** environment. You can also execute the `bin/rails 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 `config/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 `bin/rails 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 can also use the `--expanded` option to turn on the expanded table formatting mode.

```
$ bin/rails routes --expanded

--[ Route 1 ]----------------------------------------------------
Prefix            | users
Verb              | GET
URI               | /users(.:format)
Controller#Action | users#index
--[ Route 2 ]----------------------------------------------------
```

```
Prefix            |
Verb              | POST
URI               | /users(.:format)
Controller#Action | users#create
--[ Route 3 ]----------------------------------------------------
Prefix            | new_user
Verb              | GET
URI               | /users/new(.:format)
Controller#Action | users#new
--[ Route 4 ]----------------------------------------------------
Prefix            | edit_user
Verb              | GET
URI               | /users/:id/edit(.:format)
Controller#Action | users#edit
```

You can search through your routes with the grep option: -g. This outputs any routes that partially match the URL helper method name, the HTTP verb, or the URL path.

```
$ bin/rails routes -g new_comment
$ bin/rails routes -g POST
$ bin/rails routes -g admin
```

If you only want to see the routes that map to a specific controller, there's the -c option.

```
$ bin/rails routes -c users
$ bin/rails routes -c admin/users
$ bin/rails routes -c Comments
$ bin/rails routes -c Articles::CommentsController
```

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

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`

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'
```

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',
method: :post })
```

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:
'create' })
```