Rails Form helpers

Forms in web applications are an essential interface for user input. However, form markup can quickly become tedious to write and maintain because of form control naming and their numerous attributes. Rails deals away with these complexities by providing view helpers for generating form markup. However, since they have different use-cases, developers are required to know all the differences between similar helper methods before putting them to use.

In this guide you will:

- Create search forms and similar kind of generic forms not representing any specific model in your application

- Make model-centric forms for creation and editing of specific database records

- Generate select boxes from multiple types of data

- Understand the date and time helpers Rails provides

- Learn what makes a file upload form different

- Learn some cases of building forms to external resources

- Find out where to look for complex forms

This guide is not intended to be a complete documentation of available form helpers and their arguments. Please visit the Rails API documentation for a complete reference.

1 Dealing with Basic Forms

The most basic form helper is form_tag.

```
<%= form_tag do %>
  Form contents
<% end %>
```

When called without arguments like this, it creates a <form> tag which, when submitted, will POST to the current page. For instance, assuming the current page is /home/index, the generated HTML will look like this (some line breaks added for readability):

```
<form accept-charset="UTF-8" action="/home/index" method="post">
  <div style="margin:0;padding:0">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  Form contents
</form>
```

Now, you'll notice that the HTML contains something extra: a div element with two hidden input elements inside. This div is important, because the form cannot be successfully submitted without it. The first input element with name utf8 enforces browsers to properly respect your form's character encoding and is generated for all forms whether their actions are "GET" or "POST". The second input element with name authenticity_token is a security feature of Rails called **cross-site request forgery protection**, and form helpers generate it for every non-GET form (provided that this security feature is enabled). You can read more about this in the Security Guide.

Throughout this guide, the div with the hidden input elements will be excluded from code samples for brevity.

1.1 A Generic Search Form

One of the most basic forms you see on the web is a search form. This form contains:

1. a form element with "GET" method,

2. a label for the input,

3. a text input element, and

4. a submit element.

To create this form you will use form_tag, label_tag, text_field_tag, and submit_tag, respectively. Like this:

```
<%= form_tag("/search", :method => "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/search" method="get">
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

For every form input, an ID attribute is generated from its name ("q" in the example). These IDs can be very useful for CSS styling or manipulation of form controls with JavaScript.

Besides text_field_tag and submit_tag, there is a similar helper for <u>every</u> form control in HTML.

Always use "GET" as the method for search forms. This allows users to bookmark a specific search and get back to it. More generally Rails encourages you to use the right HTTP verb for an action.

1.2   Multiple Hashes in Form Helper Calls

The form_tag helper accepts 2 arguments: the path for the action and an options hash. This hash specifies the method of form submission and HTML options such as the form element's class.

As with the link_to helper, the path argument doesn't have to be given a string; it can be a hash of URL parameters recognizable by Rails' routing mechanism, which will turn the hash into a valid URL. However, since both arguments to form_tag are hashes, you

can easily run into a problem if you would like to specify both. For instance, let's say you write this:

```
form_tag(:controller => "people", :action => "search",
        :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8"
          action="/people/search?method=get&class=nifty_form"
          method="post">'
```

Here, method and class are appended to the query string of the generated URL because you even though you mean to write two hashes, you really only specified one. So you need to tell Ruby which is which by delimiting the first hash (or both) with curly brackets. This will generate the HTML you expect:

```
form_tag({:controller => "people", :action => "search"},
         :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search"
          method="get" class="nifty_form">'
```

1.3 Helpers for Generating Form Elements

Rails provides a series of helpers for generating form elements such as checkboxes, text fields, and radio buttons. These basic helpers, with names ending in "_tag" (such as text_field_tag and check_box_tag), generate just a single <input> element. The first parameter to these is always the name of the input. When the form is submitted, the name will be passed along with the form data, and will make its way to the params hash in the controller with the value entered by the user for that field. For example, if the form contains <%= text_field_tag(:query) %>, then you would be able to get the value of this field in the controller with params[:query].

When naming inputs, Rails uses certain conventions that make it possible to submit parameters with non-scalar values such as arrays or hashes, which will also be accessible in params. You can read more about them in chapter 7 of this guide. For details on the precise usage of these helpers, please refer to the API documentation.

1.3.1 Checkboxes

Checkboxes are form controls that give the user a set of options they can enable or disable:

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

This generates the following:

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

The first parameter to check_box_tag, of course, is the name of the input. The second parameter, naturally, is the value of the input. This value will be included in the form data (and be present in params) when the checkbox is checked.

1.3.2 Radio Buttons

Radio buttons, while similar to checkboxes, are controls that specify a set of options in which they are mutually exclusive (i.e., the user can only pick one):

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

Output:

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 21</label>
```

As with check_box_tag, the second parameter to radio_button_tag is the value of the input. Because these two radio buttons share the same name (age) the user will only be able to select one, and params[:age] will contain either "child" or "adult".

Always use labels for checkbox and radio buttons. They associate text with a specific option and make it easier for users to click the inputs by expanding the clickable region.

1.4 Other Helpers of Interest

Other form controls worth mentioning are textareas, password fields, hidden fields, search fields, telephone fields, URL fields and email fields:

```
<%= text_area_tag(:message, "Hi, nice site", :size => "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
```

Output:

```
<textarea id="message" name="message" cols="24" rows="6">
  Hi, nice site
</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" size="30" type="search" />
<input id="user_phone" name="user[phone]" size="30" type="tel" />
<input id="user_homepage" size="30" name="user[homepage]" type="url" />
<input id="user_address" size="30" name="user[address]" type="email" />
```

Hidden inputs are not shown to the user but instead hold data like any textual input. Values inside them can be changed with JavaScript.

The search, telephone, URL, and email inputs are HTML5 controls. If you require your app to have a consistent experience in older browsers, you will need an HTML5 polyfill (provided by CSS and/or JavaScript). There is definitely no shortage of solutions for this, although a couple of popular tools at the moment are Modernizr and yepnope, which provide a simple way to add functionality based on the presence of detected HTML5 features.

If you're using password input fields (for any purpose), you might want to configure your application to prevent those parameters from being logged. You can learn about this in the Security Guide.

2 Dealing with Model Objects

2.1 Model Object Helpers

A particularly common task for a form is editing or creating a model object. While the *_tag helpers can certainly be used for this task they are somewhat verbose as for each tag you would have to ensure the correct parameter name is used and set the default value of the input appropriately. Rails provides helpers tailored to this task. These helpers lack the _tag suffix, for example text_field, text_area.

For these helpers the first argument is the name of an instance variable and the second is the name of a method (usually an attribute) to call on that object. Rails will set the value of the input control to the return value of that method for the object and set an appropriate input name. If your controller has defined @person and that person's name is Henry then a form containing:

```
<%= text_field(:person, :name) %>
```

will produce output similar to

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

Upon form submission the value entered by the user will be stored in params[:person][:name]. The params[:person] hash is suitable for passing to Person.new or, if @person is an instance of Person, @person.update_attributes. While the name of an attribute is the most common second parameter to these helpers this is not compulsory. In the example above, as long as person objects have a name and a name= method Rails will be happy.

You must pass the name of an instance variable, i.e. :person or "person", not an actual instance of your model object.

Rails provides helpers for displaying the validation errors associated with a model object. These are covered in detail by the Active Record Validations and Callbacks guide.

2.2 Binding a Form to an Object

While this is an increase in comfort it is far from perfect. If Person has many attributes to edit then we would be repeating the name of the edited object many times. What we want to do is somehow bind a form to a model object, which is exactly what form_for does.

Assume we have a controller for dealing with articles app/controllers/articles_controller.rb:

```
def new
  @article = Article.new
end
```

The corresponding view app/views/articles/new.html.erb using form_for looks like this:

```
<%= form_for @article, :url => { :action => "create" },
    :html => {:class => "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, :size => "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

There are a few things to note here:

1. @article is the actual object being edited.

2. There is a single hash of options. Routing options are passed in the :url hash, HTML options are passed in the :html hash. Also you can provide a :namespace option for your form to ensure uniqueness of id attributes on form elements. The namespace attribute will be prefixed with underscore on the generated HTML id.

3. The form_for method yields a **form builder** object (the f variable).

4. Methods to create form controls are called **on** the form builder object f

The resulting HTML is:

```
<form accept-charset="UTF-8" action="/articles/create"
                    method="post" class="nifty_form">
<input id="article_title" name="article[title]" size="30" type="text" />
<textarea id="article_body" name="article[body]" cols="60" rows="12">
</textarea>
<input name="commit" type="submit" value="Create" />
</form>
```

The name passed to form_for controls the key used in params to access the form's values. Here the name is article and so all the inputs have names of the form article[attribute_name]. Accordingly, in the create action params[:article] will be a hash with keys :title and :body. You can read more about the significance of input names in the parameter_names section.

The helper methods called on the form builder are identical to the model object helpers except that it is not necessary to specify which object is being edited since this is already managed by the form builder.

You can create a similar binding without actually creating <form> tags with the fields_for helper. This is useful for editing additional model objects with the same form. For example if you had a Person model with an associated ContactDetail model you could create a form for creating both like so:

9

```erb
<%= form_for @person, :url => { :action => "create" } do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |contact_details_form| %>
    <%= contact_details_form.text_field :phone_number %>
  <% end %>
<% end %>
```

which produces the following output:

```html
<form accept-charset="UTF-8" action="/people/create"
    class="new_person" id="new_person" method="post">
  <input id="person_name" name="person[name]" size="30" type="text" />
  <input id="contact_detail_phone_number"
      name="contact_detail[phone_number]" size="30" type="text" />
</form>
```

The object yielded by fields_for is a form builder like the one yielded by form_for (in fact form_for calls fields_for internally).

2.3   Relying on Record Identification

The Article model is directly available to users of the application, so — following the best practices for developing with Rails — you should declare it **a resource**:

```ruby
resources :articles
```

Declaring a resource has a number of side-affects. See Rails Routing From the Outside In for more information on setting up and using resources.

When dealing with RESTful resources, calls to form_for can get significantly easier if you rely on **record identification**. In short, you can just pass the model instance and have Rails figure out model name and the rest:

```ruby
## Creating a new article
# long-style:
form_for(@article, :url => articles_path)
# same thing, short-style (record identification gets used):
```

```
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, :url => article_path(@article),
                   :html => { :method => "put" })
# short-style:
form_for(@article)
```

Notice how the short-style form_for invocation is conveniently the same, regardless of the record being new or existing. Record identification is smart enough to figure out if the record is new by asking record.new_record?. It also selects the correct path to submit to and the name based on the class of the object.

Rails will also automatically set the class and id of the form appropriately: a form creating an article would have id and class-new_article. If you were editing the article with id 23, the class would be set to edit_article and the id to edit_article_23. These attributes will be omitted for brevity in the rest of this guide.

When you're using STI (single-table inheritance) with your models, you can't rely on record identification on a subclass if only their parent class is declared a resource. You will have to specify the model name, :url, and :method explicitly.

2.3.1   Dealing with Namespaces

If you have created namespaced routes, form_for has a nifty shorthand for that too. If your application has an admin namespace then

```
form_for [:admin, @article]
```

will create a form that submits to the articles controller inside the admin namespace (submitting to admin_article_path(@article) in the case of an update). If you have several levels of namespacing then the syntax is similar:

```
form_for [:admin, :management, @article]
```

For more information on Rails' routing system and the associated conventions, please see the routing guide.

2.4   How do forms with PUT or DELETE methods work?

The Rails framework encourages RESTful design of your applications, which means you'll be making a lot of "PUT" and "DELETE" requests (besides "GET" and "POST"). However, most browsers <u>don't support</u> methods other than "GET" and "POST" when it comes to submitting forms.

Rails works around this issue by emulating other methods over POST with a hidden input named "_method", which is set to reflect the desired method:

```
form_tag(search_path, :method => "put")
```

output:

```
<form accept-charset="UTF-8" action="/search" method="post">
  <div style="margin:0;padding:0">
    <input name="_method" type="hidden" value="put" />
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  ...
```

When parsing POSTed data, Rails will take into account the special _method parameter and acts as if the HTTP method was the one specified inside it ("PUT" in this example).

3   Making Select Boxes with Ease

Select boxes in HTML require a significant amount of markup (one OPTION element for each option to choose from), therefore it makes the most sense for them to be dynamically generated.

Here is what the markup might look like:

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

Here you have a list of cities whose names are presented to the user. Internally the application only wants to handle their IDs so they are used as the options' value attribute. Let's see how Rails can help out here.

3.1 The Select and Option Tags

The most generic helper is select_tag, which — as the name implies — simply generates the SELECT tag that encapsulates an options string:

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

This is a start, but it doesn't dynamically create the option tags. You can generate option tags with the options_for_select helper:

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...]) %>
```

output:

```
<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...
```

The first argument to options_for_select is a nested array where each element has two elements: option text (city name) and option value (city id). The option value is what will be submitted to your controller. Often this will be the id of a corresponding database object but this does not have to be the case.

Knowing this, you can combine select_tag and options_for_select to achieve the desired, complete markup:

```erb
<%= select_tag(:city_id, options_for_select(...)) %>
```

options_for_select allows you to pre-select an option by passing its value.

```erb
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...], 2) %>
```

output:

```html
<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

Whenever Rails sees that the internal value of an option being generated matches this value, it will add the selected attribute to that option.

The second argument to options_for_select must be exactly equal to the desired internal value. In particular if the value is the integer 2 you cannot pass "2" to options_for_select — you must pass 2. Be aware of values extracted from the params hash as they are all strings.

3.2   Select Boxes for Dealing with Models

In most cases form controls will be tied to a specific database model and as you might expect Rails provides helpers tailored for that purpose. Consistent with other form helpers, when dealing with models you drop the _tag suffix from select_tag:

```ruby
# controller:
@person = Person.new(:city_id => 2)

# view:
<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 2], ...]) %>
```

Notice that the third parameter, the options array, is the same kind of argument you pass to options_for_select. One advantage here

is that you don't have to worry about pre-selecting the correct city if the user already has one — Rails will do this for you by reading from the @person.city_id attribute.

As with other helpers, if you were to use the select helper on a form builder scoped to the @person object, the syntax would be:

```
# select on a form builder
<%= f.select(:city_id, ...) %>
```

If you are using select (or similar helpers such as collection_select, select_tag) to set a belongs_to association you must pass the name of the foreign key (in the example above city_id), not the name of association itself. If you specify city instead of city_id Active Record will raise an error along the lines of ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750) when you pass the params hash to Person.new or update_attributes. Another way of looking at this is that form helpers only edit attributes. You should also be aware of the potential security ramifications of allowing users to edit foreign keys directly. You may wish to consider the use of attr_protected and attr_accessible. For further details on this, see the Ruby On Rails Security Guide.

3.3 Option Tags from a Collection of Arbitrary Objects

Generating options tags with options_for_select requires that you create an array containing the text and value for each option. But what if you had a City model (perhaps an Active Record one) and you wanted to generate option tags from a collection of those objects? One solution would be to make a nested array by iterating over them:

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>
<%= options_for_select(cities_array) %>
```

This is a perfectly valid solution, but Rails provides a less verbose alternative: options_from_collection_for_select. This helper expects a collection of arbitrary objects and two additional arguments:

the names of the methods to read the option **value** and **text** from, respectively:

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

As the name implies, this only generates option tags. To generate a working select box you would need to use it in conjunction with select_tag, just as you would with options_for_select. When working with model objects, just as select combines select_tag and options_for_select, collection_select combines select_tag with options_from_collection_for_select.

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

To recap, options_from_collection_for_select is to collection_select what options_for_select is to select.

Pairs passed to options_for_select should have the name first and the id second, however with options_from_collection_for_select the first argument is the value method and the second the text method.

3.4   Time Zone and Country Select

To leverage time zone support in Rails, you have to ask your users what time zone they are in. Doing so would require generating select options from a list of pre-defined TimeZone objects using collection_select, but you can simply use the time_zone_select helper that already wraps this:

```
<%= time_zone_select(:person, :time_zone) %>
```

There is also time_zone_options_for_select helper for a more manual (therefore more customizable) way of doing this. Read the API documentation to learn about the possible arguments for these two methods.

Rails used to have a country_select helper for choosing countries, but this has been extracted to the country_select plugin. When using this, be aware that the exclusion or inclusion of certain names from the list can be somewhat controversial (and was the reason this functionality was extracted from Rails).

4 Using Date and Time Form Helpers

The date and time helpers differ from all the other form helpers in two important respects:

1. Dates and times are not representable by a single input element. Instead you have several, one for each component (year, month, day etc.) and so there is no single value in your params hash with your date or time.

2. Other helpers use the _tag suffix to indicate whether a helper is a barebones helper or one that operates on model objects. With dates and times, select_date, select_time and select_datetime are the barebones helpers, date_select, time_select and datetime_select are the equivalent model object helpers.

Both of these families of helpers will create a series of select boxes for the different components (year, month, day etc.).

4.1 Barebones Helpers

The select_* family of helpers take as their first argument an instance of Date, Time or DateTime that is used as the currently selected value. You may omit this parameter, in which case the current date is used. For example

```
<%= select_date Date.today, :prefix => :start_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="start_date_year" name="start_date[year]"> ... </select>
<select id="start_date_month" name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ... </select>
```

The above inputs would result in params[:start_date] being a hash with keys :year, :month, :day. To get an actual Time or Date object you would have to extract these values and pass them to the appropriate constructor, for example

```
Date.civil(params[:start_date][:year].to_i,
           params[:start_date][:month].to_i,
           params[:start_date][:day].to_i)
```

The :prefix option is the key used to retrieve the hash of date components from the params hash. Here it was set to start_date, if omitted it will default to date.

4.2 Model Object Helpers

select_date does not work well with forms that update or create Active Record objects as Active Record expects each element of the params hash to correspond to one attribute. The model object helpers for dates and times submit parameters with special names, when Active Record sees parameters with such names it knows they must be combined with the other parameters and given to a constructor appropriate to the column type. For example:

```
<%= date_select :person, :birth_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="person_birth_date_1i" name="person[birth_date(1i)]">
</select>
<select id="person_birth_date_2i" name="person[birth_date(2i)]">
</select>
<select id="person_birth_date_3i" name="person[birth_date(3i)]">
</select>
```

which results in a params hash like

```
{:person => {'birth_date(1i)' => '2008',
'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}
```

When this is passed to Person.new (or update_attributes), Active Record spots that these parameters should all be used to construct the birth_date attribute and uses the suffixed information to determine in which order it should pass these parameters to functions such as Date.civil.

4.3 Common Options

Both families of helpers use the same core set of functions to generate the individual select tags and so both accept largely the same options. In particular, by default Rails will generate year options 5 years either side of the current year. If this is not an appropriate range, the :start_year and :end_year options override this. For an exhaustive list of the available options, refer to the API documentation.

As a rule of thumb you should be using date_select when working with model objects and select_date in other cases, such as a search form which filters results by date.

In many cases the built-in date pickers are clumsy as they do not aid the user in working out the relationship between the date and the day of the week.

4.4 Individual Components

Occasionally you need to display just a single date component such as a year or a month. Rails provides a series of helpers for this, one for each component select_year, select_month, select_day, select_hour, select_minute, select_second. These helpers are fairly straightforward. By default they will generate an input field named after the time component (for example "year" for select_year, "month" for select_month etc.) although this can be overridden with the :field_name option. The :prefix option works in the same way that it does for select_date and select_time and has the same default value.

The first parameter specifies which value should be selected and can either be an instance of a Date, Time or DateTime, in which case the relevant component will be extracted, or a numerical value. For example

```
<%= select_year(2009) %>
<%= select_year(Time.now) %>
```

will produce the same output if the current year is 2009 and the value chosen by the user can be retrieved by params[:date][:year].

5   Uploading Files

A common task is uploading some sort of file, whether it's a picture of a person or a CSV file containing data to process. The most important thing to remember with file uploads is that the rendered form's encoding **MUST** be set to "multipart/form-data". If you use form_for, this is done automatically. If you use form_tag, you must set it yourself, as per the following example.

The following two forms both upload a file.

```
<%= form_tag({:action => :upload}, :multipart => true) do %>
  <%= file_field_tag 'picture' %>
<% end %>

<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Since Rails 3.1, forms rendered using form_for have their encoding set to multipart/form-data automatically once a file_field is used inside the block. Previous versions required you to set this explicitly.

Rails provides the usual pair of helpers: the barebones file_field_tag and the model oriented file_field. The only difference with other helpers is that you cannot set a default value for file inputs as this would have no meaning. As you would expect in the first case the uploaded file is in params[:picture] and in the second case in params[:person][:picture].

5.1   What Gets Uploaded

The object in the params hash is an instance of a subclass of IO. Depending on the size of the uploaded file it may in fact be a StringIO

or an instance of File backed by a temporary file. In both cases the object will have an original_filename attribute containing the name the file had on the user's computer and a content_type attribute containing the MIME type of the uploaded file. The following snippet saves the uploaded content in #{Rails.rootpublic/uploads under the same name as the original file (assuming the form was the one in the previous example).

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads',
      uploaded_io.original_filename), 'w') do |file|
    file.write(uploaded_io.read)
  end
end
```

Once a file has been uploaded, there are a multitude of potential tasks, ranging from where to store the files (on disk, Amazon S3, etc) and associating them with models to resizing image files and generating thumbnails. The intricacies of this are beyond the scope of this guide, but there are several libraries designed to assist with these. Two of the better known ones are CarrierWave and Paperclip.

If the user has not selected a file the corresponding parameter will be an empty string.

5.2   Dealing with Ajax

Unlike other forms making an asynchronous file upload form is not as simple as providing form_for with :remote => true. With an Ajax form the serialization is done by JavaScript running inside the browser and since JavaScript cannot read files from your hard drive the file cannot be uploaded. The most common workaround is to use an invisible iframe that serves as the target for the form submission.

6 Customizing Form Builders

As mentioned previously the object yielded by form_for and fields_for
is an instance of FormBuilder (or a subclass thereof). Form builders
encapsulate the notion of displaying form elements for a single ob-
ject. While you can of course write helpers for your forms in the
usual way you can also subclass FormBuilder and add the helpers
there. For example

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

can be replaced with

```
<%= form_for @person, :builder => LabellingFormBuilder do |f| %>
  <%= f.text_field :first_name %>
<% end %>
```

by defining a LabellingFormBuilder class similar to the follow-
ing:

```
class LabellingFormBuilder < ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

If you reuse this frequently you could define a labeled_form_for
helper that automatically applies the :builder => LabellingForm-
Builder option.

The form builder used also determines what happens when you
do

```
<%= render :partial => f %>
```

If f is an instance of FormBuilder then this will render the form
partial, setting the partial's object to the form builder. If the form
builder is of class LabellingFormBuilder then the labelling_form par-
tial would be rendered instead.

7 Understanding Parameter Naming Conventions

As you've seen in the previous sections, values from forms can be at the top level of the params hash or nested in another hash. For example in a standard create action for a Person model, params[:model] would usually be a hash of all the attributes for the person to create. The params hash can also contain arrays, arrays of hashes and so on.

Fundamentally HTML forms don't know about any sort of structured data, all they generate is name–value pairs, where pairs are just plain strings. The arrays and hashes you see in your application are the result of some parameter naming conventions that Rails uses.

You may find you can try out examples in this section faster by using the console to directly invoke Racks' parameter parser. For example,

```
Rack::Utils.parse_query "name=fred&phone=0123456789"
# => {"name"=>"fred", "phone"=>"0123456789"}
```

7.1 Basic Structures

The two basic structures are arrays and hashes. Hashes mirror the syntax used for accessing the value in params. For example if a form contains

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

the params hash will contain

```
{'person' => {'name' => 'Henry'}}
```

and params[:person][:name] will retrieve the submitted value in the controller.

Hashes can be nested as many levels as required, for example

```
<input id="person_address_city" name="person[address][city]"
                        type="text" value="New York"/>
```

will result in the params hash being

```
{'person' => {'address' => {'city' => 'New York'}}}
```

Normally Rails ignores duplicate parameter names. If the parameter name contains an empty set of square brackets [] then they will be accumulated in an array. If you wanted people to be able to input multiple phone numbers, you could place this in the form:

```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```

This would result in params[:person][:phone_number] being an array.

7.2   Combining Them

We can mix and match these two concepts. For example, one element of a hash might be an array as in the previous example, or you can have an array of hashes. For example a form might let you create any number of addresses by repeating the following form fragment

```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```

This would result in params[:addresses] being an array of hashes with keys line1, line2 and city. Rails decides to start accumulating values in a new hash whenever it encounters an input name that already exists in the current hash.

There's a restriction, however, while hashes can be nested arbitrarily, only one level of "arrayness" is allowed. Arrays can be usually replaced by hashes, for example instead of having an array of model objects one can have a hash of model objects keyed by their id, an array index or some other parameter.

Array parameters do not play well with the check_box helper. According to the HTML specification unchecked checkboxes submit no value. However it is often convenient for a checkbox to always submit a value. The check_box helper fakes this by creating an auxiliary hidden input with the same name. If the checkbox is unchecked only the hidden input is submitted and if it is checked then both are submitted but the value submitted by the checkbox takes precedence. When working with array parameters this duplicate submission will confuse Rails since duplicate input names are how it decides when to start a new array element. It is preferable to either use check_box_tag or to use hashes instead of arrays.

7.3   Using Form Helpers

The previous sections did not use the Rails form helpers at all. While you can craft the input names yourself and pass them directly to helpers such as text_field_tag Rails also provides higher level support. The two tools at your disposal here are the name parameter to form_for and fields_for and the :index option that helpers take.

You might want to render a form with a set of edit fields for each of a person's addresses. For example:

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
  <%= person_form.fields_for address, :index => address do |address_form|%>
      <%= address_form.text_field :city %>
  <% end %>
  <% end %>
<% end %>
```

Assuming the person had two addresses, with ids 23 and 45 this would create output similar to this:

```
<form accept-charset="UTF-8" action="/people/1"
      class="edit_person" id="edit_person_1" method="post">
<input id="person_name" name="person[name]" size="30" type="text" />
```

```
<input id="person_address_23_city"
  name="person[address][23][city]" size="30" type="text" />
<input id="person_address_45_city"
  name="person[address][45][city]" size="30" type="text" />
</form>
```

This will result in a params hash that looks like

```
{'person' => {'name' => 'Bob', 'address' =>
    {'23' => {'city' => 'Paris'}, '45' => {'city' => 'London'}}}}
```

Rails knows that all these inputs should be part of the person hash because you called fields_for on the first form builder. By specifying an :index option you're telling Rails that instead of naming the inputs person[address][city] it should insert that index surrounded by [] between the address and the city. If you pass an Active Record object as we did then Rails will call to_param on it, which by default returns the database id. This is often useful as it is then easy to locate which Address record should be modified. You can pass numbers with some other significance, strings or even nil (which will result in an array parameter being created).

To create more intricate nestings, you can specify the first part of the input name (person[address] in the previous example) explicitly, for example

```
<%= fields_for 'person[address][primary]', address,
               :index => address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

will create inputs like

```
<input id="person_address_primary_1_city"
       name="person[address][primary][1][city]"
       size="30" type="text" value="bologna" />
```

As a general rule the final input name is the concatenation of the name given to fields_for/form_for, the index value and the name of

```

the attribute. You can also pass an :index option directly to helpers such as text_field, but it is usually less repetitive to specify this at the form builder level rather than on individual input controls.

As a shortcut you can append [] to the name and omit the :index option. This is the same as specifying :index => address so

```
<%= fields_for 'person[address][primary][]', address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

produces exactly the same output as the previous example.

## 8    Forms to external resources

If you need to post some data to an external resource it is still great to build your from using rails form helpers. But sometimes you need to set an authenticity_token for this resource. You can do it by passing an :authenticity_token => 'your_external_token' parameter to the form_tag options:

```
<%= form_tag 'http://farfar.away/form',
    :authenticity_token => 'external_token') do %>
  Form contents
<% end %>
```

Sometimes when you submit data to an external resource, like payment gateway, fields you can use in your form are limited by an external API. So you may want not to generate an authenticity_token hidden field at all. For doing this just pass false to the :authenticity_token option:

```
<%= form_tag 'http://farfar.away/form',
    :authenticity_token => false) do %>
  Form contents
<% end %>
```

The same technique is available for the form_for too:

```
<%= form_for @invoice, :url => external_url,
    :authenticity_token => 'external_token' do |f|
  Form contents
<% end %>
```

Or if you don't want to render an authenticity_token field:

```
<%= form_for @invoice, :url => external_url,
    :authenticity_token => false do |f|
  Form contents
<% end %>
```

9   Building Complex Forms

Many apps grow beyond simple forms editing a single object. For example when creating a Person you might want to allow the user to (on the same form) create multiple address records (home, work, etc.). When later editing that person the user should be able to add, remove or amend addresses as necessary. While this guide has shown you all the pieces necessary to handle this, Rails does not yet have a standard end-to-end way of accomplishing this, but many have come up with viable approaches. These include:

- As of Rails 2.3, Rails includes Nested Attributes and Nested Object Forms

- Ryan Bates' series of Railscasts on complex forms

- Handle Multiple Models in One Form from Advanced Rails Recipes

- Eloy Duran's complex-forms-examples application

- Lance Ivy's nested_assignment plugin and sample application

- James Golick's attribute_fu plugin

```

# Contents