DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Action View 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 the need to handle form control naming and its numerous attributes. Rails does away with this complexity by providing view helpers for generating form markup. However, since these helpers have different use cases, developers need to know the differences between the helper methods before putting them to use.

After reading this guide, you will know:

- How to create search forms and similar kind of generic forms not representing any specific model in your application.
- How to make model-centric forms for creating and editing specific database records.
- How to generate select boxes from multiple types of data.
- What date and time helpers Rails provides.
- What makes a file upload form different.
- How to post forms to external resources and specify setting an `authenticity_token`.
- How to build complex forms.

NOTE: 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.

Dealing with Basic Forms

The main form helper is `form_with`.

```
<%= form_with do |form| %>
  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 a home page, the generated HTML will look like this:

```
<form accept-charset="UTF-8" action="/" method="post">
  <input name="authenticity_token" type="hidden" value="J7CBxfHalt49OSHp27hblqK20c9PgwJ108nI
  Form contents
</form>
```

You'll notice that the HTML contains an `input` element with type `hidden`. This `input` is important, because non-GET forms cannot be successfully submitted without it. The hidden input element with the 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 Securing Rails Applications guide.

A Generic Search Form

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

- a form element with "GET" method,
- a label for the input,
- a text input element, and
- a submit element.

To create this form you will use `form_with` and the form builder object it yields. Like so:

```
<%= form_with url: "/search", method: :get do |form| %>
  <%= form.label :query, "Search for:" %>
  <%= form.text_field :query %>
  <%= form.submit "Search" %>
<% end %>
```

This will generate the following HTML:

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

TIP: Passing `url: my_specified_path` to `form_with` tells the form where to make the request. However, as explained below, you can also pass ActiveRecord objects to the form.

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

IMPORTANT: 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.

Helpers for Generating Form Elements

The form builder object yielded by `form_with` provides numerous helper methods for generating form elements such as text fields, checkboxes, and radio buttons. The first parameter to these methods 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` in the controller with the value entered by the user for that field. For example, if the form contains `<%= form.text_field :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 Understanding Parameter Naming Conventions of this guide. For details on the precise usage of these helpers, please refer to the API documentation.

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

```
<%= form.check_box :pet_dog %>
<%= form.label :pet_dog, "I own a dog" %>
<%= form.check_box :pet_cat %>
<%= form.label :pet_cat, "I own a cat" %>
```

This generates the following:

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

The first parameter to `check_box` is the name of the input. The second parameter 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.

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):

```
<%= form.radio_button :age, "child" %>
<%= form.label :age_child, "I am younger than 21" %>
<%= form.radio_button :age, "adult" %>
<%= form.label :age_adult, "I am over 21" %>
```

Output:

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

```
<input type="radio" id="age_adult" name="age" value="adult" />
<label for="age_adult">I am over 21</label>
```

As with `check_box`, the second parameter to `radio_button` 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 of them, and `params[:age]` will contain either `"child"` or `"adult"`.

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

Other Helpers of Interest

Other form controls worth mentioning are text areas, hidden fields, password fields, number fields, date and time fields, and many more:

```
<%= form.text_area :message, size: "70x5" %>
<%= form.hidden_field :parent_id, value: "foo" %>
<%= form.password_field :password %>
<%= form.number_field :price, in: 1.0..20.0, step: 0.5 %>
<%= form.range_field :discount, in: 1..100 %>
<%= form.date_field :born_on %>
<%= form.time_field :started_at %>
<%= form.datetime_local_field :graduation_day %>
<%= form.month_field :birthday_month %>
<%= form.week_field :birthday_week %>
<%= form.search_field :name %>
<%= form.email_field :address %>
<%= form.telephone_field :phone %>
<%= form.url_field :homepage %>
<%= form.color_field :favorite_color %>
```

Output:

```
<textarea name="message" id="message" cols="70" rows="5"></textarea>
<input type="hidden" name="parent_id" id="parent_id" value="foo" />
<input type="password" name="password" id="password" />
<input type="number" name="price" id="price" step="0.5" min="1.0" max="20.0" />
<input type="range" name="discount" id="discount" min="1" max="100" />
<input type="date" name="born_on" id="born_on" />
<input type="time" name="started_at" id="started_at" />
<input type="datetime-local" name="graduation_day" id="graduation_day" />
<input type="month" name="birthday_month" id="birthday_month" />
<input type="week" name="birthday_week" id="birthday_week" />
<input type="search" name="name" id="name" />
<input type="email" name="address" id="address" />
<input type="tel" name="phone" id="phone" />
```

```
<input type="url" name="homepage" id="homepage" />
<input type="color" name="favorite_color" id="favorite_color" value="#000000" />
```

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

IMPORTANT: The search, telephone, date, time, color, datetime, datetime-local, month, week, URL, email, number, and range 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 popular tool at the moment is Modernizr, which provides a simple way to add functionality based on the presence of detected HTML5 features.

TIP: 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 Securing Rails Applications guide.

Dealing with Model Objects

Binding a Form to an Object

The `:model` argument of `form_with` allows us to bind the form builder object to a model object. This means that the form will be scoped to that model object, and the form's fields will be populated with values from that model object.

For example, if we have an `@article` model object like:

```
@article = Article.find(42)
# => #<Article id: 42, title: "My Title", body: "My Body">
```

The following form:

```
<%= form_with model: @article do |form| %>
  <%= form.text_field :title %>
  <%= form.text_area :body, size: "60x10" %>
  <%= form.submit %>
<% end %>
```

Outputs:

```
<form action="/articles/42" method="post" accept-charset="UTF-8" >
  <input name="authenticity_token" type="hidden" value="..." />
  <input type="text" name="article[title]" id="article_title" value="My Title" />
  <textarea name="article[body]" id="article_body" cols="60" rows="10">
    My Body
  </textarea>
  <input type="submit" name="commit" value="Update Article" data-disable-with="Update Articl
</form>
```

There are several things to notice here:

- The form `action` is automatically filled with an appropriate value for `@article`.
- The form fields are automatically filled with the corresponding values from `@article`.
- The form field names are scoped with `article[...]`. This means that `params[:article]` will be a hash containing all these field's values. You can read more about the significance of input names in chapter Understanding Parameter Naming Conventions of this guide.
- The submit button is automatically given an appropriate text value.

TIP: Conventionally your inputs will mirror model attributes. However, they don't have to! If there is other information you need you can include it in your form just as with attributes and access it via `params[:article][:my_nifty_non_attribute_input]`.

The `fields_for` Helper The `fields_for` helper creates a similar binding but without rendering a `<form>` tag. This can be used to render fields for additional model objects within the same form. For example, if you had a `Person` model with an associated `ContactDetail` model, you could create a single form for both like so:

```
<%= form_with model: @person do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for :contact_detail, @person.contact_detail do |contact_detail_form| %>
    <%= contact_detail_form.text_field :phone_number %>
  <% end %>
<% end %>
```

Which produces the following output:

```
<form action="/people" accept-charset="UTF-8" method="post">
  <input type="hidden" name="authenticity_token" value="bL13x72pldyDD8bgtkjKQakJCpd4A8JdXGb
  <input type="text" name="person[name]" id="person_name" />
  <input type="text" name="contact_detail[phone_number]" id="contact_detail_phone_number" />
</form>
```

The object yielded by `fields_for` is a form builder like the one yielded by `form_with`.

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**:

```
resources :articles
```

TIP: Declaring a resource has a number of side effects. See Rails Routing from the Outside In guide for more information on setting up and using resources.

When dealing with RESTful resources, calls to `form_with` 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. In both of these examples, the long and short style have the same outcome:

```
## Creating a new article
# long-style:
form_with(model: @article, url: articles_path)
# short-style:
form_with(model: @article)

## Editing an existing article
# long-style:
form_with(model: @article, url: article_path(@article), method: "patch")
# short-style:
form_with(model: @article)
```

Notice how the short-style `form_with` 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.persisted?`. It also selects the correct path to submit to, and the name based on the class of the object.

If you have a singular resource, you will need to call `resource` and `resolve` for it to work with `form_with`:

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

WARNING: 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 `:url`, and `:scope` (the model name) explicitly.

Dealing with Namespaces If you have created namespaced routes, `form_with` has a nifty shorthand for that too. If your application has an admin namespace then

```
form_with model: [:admin, @article]
```

will create a form that submits to the `ArticlesController` 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_with model: [:admin, :management, @article]
```

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

How do forms with PATCH, PUT, or DELETE methods work?

The Rails framework encourages RESTful design of your applications, which means you'll be making a lot of "PATCH", "PUT", and "DELETE" requests (besides "GET" and "POST"). However, most browsers *don't support* 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_with(url: search_path, method: "patch")
```

Output:

```
<form accept-charset="UTF-8" action="/search" method="post">
  <input name="_method" type="hidden" value="patch" />
  <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf
  <!-- ... -->
</form>
```

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

When rendering a form, submission buttons can override the declared `method` attribute through the `formmethod:` keyword:

```
<%= form_with url: "/posts/1", method: :patch do |form| %>
  <%= form.button "Delete", formmethod: :delete, data: { confirm: "Are you sure?" } %>
  <%= form.button "Update" %>
<% end %>
```

Similar to `<form>` elements, most browsers *don't support* overriding form methods declared through formmethod other than "GET" and "POST".

Rails works around this issue by emulating other methods over POST through a combination of formmethod, value, and name attributes:

```
<form accept-charset="UTF-8" action="/posts/1" method="post">
  <input name="_method" type="hidden" value="patch" />
  <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf
  <!-- ... -->

  <button type="submit" formmethod="post" name="_method" value="delete" data-confirm="Are yo
  <button type="submit" name="button">Update</button>
</form>
```

IMPORTANT: In Rails 6.0 and 5.2, all forms using `form_with` implement `remote: true` by default. These forms will submit data using an XHR (Ajax) request. To disable this include `local: true`. To dive deeper see Working with JavaScript in Rails guide.

Making Select Boxes with Ease

Select boxes in HTML require a significant amount of markup - one `<option>` element for each option to choose from. So Rails provides helper methods to reduce this burden.

For example, let's say we have a list of cities for the user to choose from. We can use the `select` helper like so:

```
<%= form.select :city, ["Berlin", "Chicago", "Madrid"] %>
```

Output:

```
<select name="city" id="city">
  <option value="Berlin">Berlin</option>
  <option value="Chicago">Chicago</option>
  <option value="Madrid">Madrid</option>
</select>
```

We can also designate `<option>` values that differ from their labels:

```
<%= form.select :city, [["Berlin", "BE"], ["Chicago", "CHI"], ["Madrid", "MD"]] %>
```

Output:

```
<select name="city" id="city">
  <option value="BE">Berlin</option>
  <option value="CHI">Chicago</option>
  <option value="MD">Madrid</option>
</select>
```

This way, the user will see the full city name, but `params[:city]` will be one of `"BE"`, `"CHI"`, or `"MD"`.

Lastly, we can specify a default choice for the select box with the `:selected` argument:

```
<%= form.select :city, [["Berlin", "BE"], ["Chicago", "CHI"], ["Madrid", "MD"]], selected: '
```

Output:

```
<select name="city" id="city">
  <option value="BE">Berlin</option>
  <option value="CHI" selected="selected">Chicago</option>
  <option value="MD">Madrid</option>
</select>
```

Option Groups

In some cases we may want to improve the user experience by grouping related options together. We can do so by passing a `Hash` (or comparable `Array`) to `select`:

```
<%= form.select :city,
      {
        "Europe" => [ ["Berlin", "BE"], ["Madrid", "MD"] ],
        "North America" => [ ["Chicago", "CHI"] ],
      },
      selected: "CHI" %>
```

Output:

```
<select name="city" id="city">
  <optgroup label="Europe">
    <option value="BE">Berlin</option>
    <option value="MD">Madrid</option>
  </optgroup>
  <optgroup label="North America">
    <option value="CHI" selected="selected">Chicago</option>
  </optgroup>
</select>
```

Select Boxes and Model Objects

Like other form controls, a select box can be bound to a model attribute. For example, if we have a `@person` model object like:

```
@person = Person.new(city: "MD")
```

The following form:

```
<%= form_with model: @person do |form| %>
  <%= form.select :city, [["Berlin", "BE"], ["Chicago", "CHI"], ["Madrid", "MD"]] %>
<% end %>
```

Outputs a select box like:

```
<select name="person[city]" id="person_city">
  <option value="BE">Berlin</option>
  <option value="CHI">Chicago</option>
  <option value="MD" selected="selected">Madrid</option>
</select>
```

Notice that the appropriate option was automatically marked `selected="selected"`. Since this select box was bound to a model, we didn't need to specify a `:selected` argument!

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 `ActiveSupport::TimeZone` objects, but you can simply use the `time_zone_select` helper that already wraps this:

```
<%= form.time_zone_select :time_zone %>
```

Rails *used* to have a `country_select` helper for choosing countries, but this has been extracted to the country_select plugin.

Using Date and Time Form Helpers

If you do not wish to use HTML5 date and time inputs, Rails provides alternative date and time form helpers that render plain select boxes. These helpers render a select box for each temporal component (e.g. year, month, day, etc). For example, if we have a `@person` model object like:

```
@person = Person.new(birth_date: Date.new(1995, 12, 21))
```

The following form:

```
<%= form_with model: @person do |form| %>
  <%= form.date_select :birth_date %>
<% end %>
```

Outputs select boxes like:

```
<select name="person[birth_date(1i)]" id="person_birth_date_1i">
  <option value="1990">1990</option>
  <option value="1991">1991</option>
  <option value="1992">1992</option>
  <option value="1993">1993</option>
  <option value="1994">1994</option>
  <option value="1995" selected="selected">1995</option>
  <option value="1996">1996</option>
  <option value="1997">1997</option>
  <option value="1998">1998</option>
  <option value="1999">1999</option>
  <option value="2000">2000</option>
</select>
<select name="person[birth_date(2i)]" id="person_birth_date_2i">
  <option value="1">January</option>
  <option value="2">February</option>
  <option value="3">March</option>
  <option value="4">April</option>
  <option value="5">May</option>
  <option value="6">June</option>
  <option value="7">July</option>
  <option value="8">August</option>
  <option value="9">September</option>
  <option value="10">October</option>
  <option value="11">November</option>
  <option value="12" selected="selected">December</option>
</select>
```

```
<select name="person[birth_date(3i)]" id="person_birth_date_3i">
  <option value="1">1</option>
  ...
  <option value="21" selected="selected">21</option>
  ...
  <option value="31">31</option>
</select>
```

Notice that, when the form is submitted, there will be no single value in the `params` hash that contains the full date. Instead, there will be several values with special names like `"birth_date(1i)"`. Active Record knows how to assemble these specially-named values into a full date or time, based on the declared type of the model attribute. So we can pass `params[:person]` to e.g. `Person.new` or `Person#update` just like we would if the form used a single field to represent the full date.

In addition to the `date_select` helper, Rails provides `time_select` and `datetime_select`.

Select Boxes for Individual Temporal Components

Rails also provides helpers to render select boxes for individual temporal components: `select_year`, `select_month`, `select_day`, `select_hour`, `select_minute`, and `select_second`. These helpers are "bare" methods, meaning they are not called on a form builder instance. For example:

```
<%= select_year 1999, prefix: "party" %>
```

Outputs a select box like:

```
<select name="party[year]" id="party_year">
  <option value="1994">1994</option>
  <option value="1995">1995</option>
  <option value="1996">1996</option>
  <option value="1997">1997</option>
  <option value="1998">1998</option>
  <option value="1999" selected="selected">1999</option>
  <option value="2000">2000</option>
  <option value="2001">2001</option>
  <option value="2002">2002</option>
  <option value="2003">2003</option>
  <option value="2004">2004</option>
</select>
```

For each of these helpers, you may specify a date or time object instead of a number as the default value, and the appropriate temporal component will be extracted and used.

Choices from a Collection of Arbitrary Objects

Often, we want to generate a set of choices in a form from a collection of objects. For example, when we want the user to choose from cities in our database, and we have a `City` model like:

```
City.order(:name).to_a
# => [
#      #<City id: 3, name: "Berlin">,
#      #<City id: 1, name: "Chicago">,
#      #<City id: 2, name: "Madrid">
#    ]
```

Rails provides helpers that generate choices from a collection without having to explicitly iterate over it. These helpers determine the value and text label of each choice by calling specified methods on each object in the collection.

The `collection_select` Helper

To generate a select box for our cities, we can use `collection_select`:

```
<%= form.collection_select :city_id, City.order(:name), :id, :name %>
```

Output:

```
<select name="city_id" id="city_id">
  <option value="3">Berlin</option>
  <option value="1">Chicago</option>
  <option value="2">Madrid</option>
</select>
```

NOTE: With `collection_select` we specify the value method first (`:id` in the example above), and the text label method second (`:name` in the example above). This is opposite of the order used when specifying choices for the `select` helper, where the text label comes first and the value second.

The `collection_radio_buttons` Helper

To generate a set of radio buttons for our cities, we can use `collection_radio_buttons`:

```
<%= form.collection_radio_buttons :city_id, City.order(:name), :id, :name %>
```

Output:

```
<input type="radio" name="city_id" value="3" id="city_id_3">
<label for="city_id_3">Berlin</label>
<input type="radio" name="city_id" value="1" id="city_id_1">
<label for="city_id_1">Chicago</label>
<input type="radio" name="city_id" value="2" id="city_id_2">
<label for="city_id_2">Madrid</label>
```

The `collection_check_boxes` Helper

To generate a set of check boxes for our cities (which allows users to choose more than one), we can use `collection_check_boxes`:

```
<%= form.collection_check_boxes :city_id, City.order(:name), :id, :name %>
```

Output:

```
<input type="checkbox" name="city_id[]" value="3" id="city_id_3">
<label for="city_id_3">Berlin</label>
<input type="checkbox" name="city_id[]" value="1" id="city_id_1">
<label for="city_id_1">Chicago</label>
<input type="checkbox" name="city_id[]" value="2" id="city_id_2">
<label for="city_id_2">Madrid</label>
```

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. File upload fields can be rendered with the `file_field` helper.

```
<%= form_with model: @person do |form| %>
  <%= form.file_field :picture %>
<% end %>
```

The most important thing to remember with file uploads is that the rendered form's `enctype` attribute **must** be set to "multipart/form-data". This is done automatically if you use a `file_field` inside a `form_with`. You can also set the attribute manually:

```
<%= form_with url: "/uploads", multipart: true do |form| %>
  <%= file_field_tag :picture %>
<% end %>
```

Note that, in accordance with `form_with` conventions, the field names in the two forms above will also differ. That is, the field name in the first form will be `person[picture]` (accessible via `params[:person][:picture]`), and the field name in the second form will be just `picture` (accessible via `params[:picture]`).

What Gets Uploaded

The object in the `params` hash is an instance of `ActionDispatch::Http::UploadedFile`. The following snippet saves the uploaded file in `#{Rails.root}/public/uploads` under the same name as the original file.

```
def upload
  uploaded_file = params[:picture]
  File.open(Rails.root.join('public', 'uploads', uploaded_file.original_filename), 'wb') do
    file.write(uploaded_file.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), associating them with models, resizing image files, and generating thumbnails, etc. Active Storage is designed to assist with these tasks.

Customizing Form Builders

The object yielded by `form_with` and `fields_for` is an instance of `ActionView::Helpers::FormBuilder`. Form builders encapsulate the notion of displaying form elements for a single object. While you can write helpers for your forms in the usual way, you can also create a subclass of `ActionView::Helpers::FormBuilder`, and add the helpers there. For example,

```
<%= form_with model: @person do |form| %>
  <%= text_field_with_label form, :first_name %>
<% end %>
```

can be replaced with

```
<%= form_with model: @person, builder: LabellingFormBuilder do |form| %>
  <%= form.text_field :first_name %>
<% end %>
```

by defining a `LabellingFormBuilder` class similar to the following:

```
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_with` helper that automatically applies the `builder: LabellingFormBuilder` option:

```
def labeled_form_with(model: nil, scope: nil, url: nil, format: nil, **options, &block)
  options.merge! builder: LabellingFormBuilder
  form_with model: model, scope: scope, url: url, format: format, **options, &block
end
```

The form builder used also determines what happens when you do:

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

If `f` is an instance of `ActionView::Helpers::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` partial would be rendered instead.

Understanding Parameter Naming Conventions

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[:person]` 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.

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 ends with an empty set of square brackets `[]` then they will be accumulated in an array. If you wanted users 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 containing the inputted phone numbers.

Combining Them

We can mix and match these two concepts. 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="person[addresses][][line1]" type="text"/>
<input name="person[addresses][][line2]" type="text"/>
<input name="person[addresses][][city]" type="text"/>
<input name="person[addresses][][line1]" type="text"/>
<input name="person[addresses][][line2]" type="text"/>
<input name="person[addresses][][city]" type="text"/>
```

This would result in `params[:person][:addresses]` being an array of hashes with keys `line1`, `line2`, and `city`.

There's a restriction, however: while hashes can be nested arbitrarily, only one level of "arrayness" is allowed. Arrays can usually be 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.

WARNING: 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.

The `fields_for` Helper `:index` Option

Let's say we want to render a form with a set of fields for each of a person's addresses. The `fields_for` helper with its `:index` option can assist:

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

Assuming the person has two addresses with IDs 23 and 45, the above form would render output similar to:

```
<form accept-charset="UTF-8" action="/people/1" method="post">
  <input name="_method" type="hidden" value="patch" />
  <input id="person_name" name="person[name]" type="text" />
  <input id="person_address_23_city" name="person[address][23][city]" type="text" />
  <input id="person_address_45_city" name="person[address][45][city]" type="text" />
</form>
```

Which will result in a `params` hash that looks like:

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

All of the form inputs map to the `"person"` hash because we called `fields_for` on the `person_form` form builder. By specifying an `:index` option, we mapped the address inputs to `person[address][#{address.id}][city]` instead of `person[address][city]`. Thus we are able to determine which Address records should be modified when processing the `params` hash.

You can pass other numbers or strings of significance via the `:index` option. You can even pass `nil`, which will produce an array parameter.

To create more intricate nestings, you can specify the leading portion of the input name explicitly. For example:

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

will create inputs like:

```
<input id="person_address_primary_23_city" name="person[address][primary][23][city]" type="t
```

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 than on individual input fields.

Speaking generally, the final input name will be a concatenation of the name given to `fields_for` / `form_with`, the `:index` option value, and the name of the attribute.

Lastly, as a shortcut, instead of specifying an ID for `:index` (e.g. `index: address.id`), you can append `"[]"` to the given name. For example:

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

produces exactly the same output as our original example.

Forms to External Resources

Rails' form helpers can also be used to build a form for posting data to an external resource. However, at times it can be necessary to set an `authenticity_token` for the resource; this can be done by passing an `authenticity_token: 'your_external_token'` parameter to the `form_with` options:

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

Sometimes when submitting data to an external resource, like a payment gateway, the fields that can be used in the form are limited by an external API and it may be undesirable to generate an `authenticity_token`. To not send a token, simply pass `false` to the `:authenticity_token` option:

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

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.

Configuring the Model

Active Record provides model level support via the `accepts_nested_attributes_for` method:

```
class Person < ApplicationRecord
  has_many :addresses, inverse_of: :person
  accepts_nested_attributes_for :addresses
end

class Address < ApplicationRecord
  belongs_to :person
end
```

This creates an `addresses_attributes=` method on `Person` that allows you to create, update, and (optionally) destroy addresses.

Nested Forms

The following form allows a user to create a `Person` and its associated addresses.

```
<%= form_with model: @person do |form| %>
  Addresses:
```

```
  <ul>
    <%= form.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>

        <%= addresses_form.label :street %>
        <%= addresses_form.text_field :street %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

When an association accepts nested attributes `fields_for` renders its block once for every element of the association. In particular, if a person has no addresses it renders nothing. A common pattern is for the controller to build one or more empty children so that at least one set of fields is shown to the user. The example below would result in 2 sets of address fields being rendered on the new person form.

```
def new
  @person = Person.new
  2.times { @person.addresses.build }
end
```

The `fields_for` yields a form builder. The parameters' name will be what `accepts_nested_attributes_for` expects. For example, when creating a user with 2 addresses, the submitted parameters would look like:

```
{
  'person' => {
    'name' => 'John Doe',
    'addresses_attributes' => {
      '0' => {
        'kind' => 'Home',
        'street' => '221b Baker Street'
      },
      '1' => {
        'kind' => 'Office',
        'street' => '31 Spooner Street'
      }
    }
  }
}
```

The keys of the `:addresses_attributes` hash are unimportant, they need merely be different for each address.

If the associated object is already saved, `fields_for` autogenerates a hidden input with the `id` of the saved record. You can disable this by passing `include_id: false` to `fields_for`.

The Controller

As usual you need to declare the permitted parameters in the controller before you pass them to the model:

```
def create
  @person = Person.new(person_params)
  # ...
end

private
  def person_params
    params.require(:person).permit(:name, addresses_attributes: [:id, :kind, :street])
  end
```

Removing Objects

You can allow users to delete associated objects by passing `allow_destroy: true` to `accepts_nested_attributes_for`

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, allow_destroy: true
end
```

If the hash of attributes for an object contains the key `_destroy` with a value that evaluates to `true` (e.g. 1, '1', true, or 'true') then the object will be destroyed. This form allows users to remove addresses:

```
<%= form_with model: @person do |form| %>
  Addresses:
  <ul>
    <%= form.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.check_box :_destroy %>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

Don't forget to update the permitted params in your controller to also include the `_destroy` field:

```ruby
def person_params
  params.require(:person).
    permit(:name, addresses_attributes: [:id, :kind, :street, :_destroy])
end
```

Preventing Empty Records

It is often useful to ignore sets of fields that the user has not filled in. You can control this by passing a `:reject_if` proc to `accepts_nested_attributes_for`. This proc will be called with each hash of attributes submitted by the form. If the proc returns `true` then Active Record will not build an associated object for that hash. The example below only tries to build an address if the `kind` attribute is set.

```ruby
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, reject_if: lambda {|attributes| attributes['kin
end
```

As a convenience you can instead pass the symbol `:all_blank` which will create a proc that will reject records where all the attributes are blank excluding any value for `_destroy`.

Adding Fields on the Fly

Rather than rendering multiple sets of fields ahead of time you may wish to add them only when a user clicks on an "Add new address" button. Rails does not provide any built-in support for this. When generating new sets of fields you must ensure the key of the associated array is unique - the current JavaScript date (milliseconds since the epoch) is a common choice.

Using Tag Helpers Without a Form Builder

In case you need to render form fields outside of the context of a form builder, Rails provides tag helpers for common form elements. For example, `check_box_tag`:

```
<%= check_box_tag "accept" %>
```

Output:

```
<input type="checkbox" name="accept" id="accept" value="1" />
```

Generally, these helpers have the same name as their form builder counterparts plus a `_tag` suffix. For a complete list, see the `FormTagHelper` API documentation.

Using `form_tag` and `form_for`

Before `form_with` was introduced in Rails 5.1 its functionality used to be split between `form_tag` and `form_for`. Both are now soft-deprecated. Documentation on their usage can be found in older versions of this guide.