SimpleForm - Rails forms made easy.

SimpleForm aims to be as flexible as possible while helping you with powerful components to create your forms. The basic goal of SimpleForm is to not touch your way of defining the layout, letting you find the better design for your eyes. Most of the DSL was inherited from Formtastic, which we are thankful for and should make you feel right at home.

INFO: This README is also available in a friendly navigable format [1] and refers to **SimpleForm** 2.0. If you are using **SimpleForm** in the versions 1.x, you should check this branch:

https://github.com/plataformatec/simple_form/tree/v1.5 [2]

Installation

Add it to your Gemfile:

```
gem 'simple_form'
```

Run the following command to install it:

```
bundle install
```

Run the generator:

```
rails generate simple_form:install
```

Also, if you want to use the country select, you will need the country_select gem [3], add it to your Gemfile:

```
gem 'country_select'
```

Twitter Bootstrap

SimpleForm 2.0 can be easily integrated to the Twitter Bootstrap [4]. To do that you have to use the bootstrap option in the install generator, like this:

```
rails generate simple_form:install --bootstrap
```

You have to be sure that you added a copy of the Twitter Bootstrap [5] assets on your application.

For more information see the generator output, our example application code [6] and the live example app [7].

NOTE: **SimpleForm** integration requires Twitter Bootstrap version 2.0 or higher.

Zurb Foundation 3

To generate wrappers that are compatible with Zurb Foundation 3 [8], pass the foundation option to the generator, like this:

```
rails generate simple_form:install --foundation
```

Please note that the Foundation wrapper does not support the :hint option by default. In order to enable hints, please uncomment the appropriate line in config/initializers/simple_form_foundation.rb. You will need to provide your own CSS styles for hints.

Please see the instructions on how to install Foundation in a Rails app [9].

Usage

SimpleForm was designed to be customized as you need to. Basically it's a stack of components that are invoked to create a complete html input for you, which by default contains label, hints, errors and the input itself. It does not aim to create a lot of different logic from the default Rails form helpers, as they do a great work by themselves. Instead, **SimpleForm** acts as a DSL and just maps your input type (retrieved from the column

definition in the database) to a specific helper method.

To start using **SimpleForm** you just have to use the helper it provides:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username %>
  <%= f.input :password %>
  <%= f.button :submit %>
<% end %>
```

This will generate an entire form with labels for user name and password as well, and render errors by default when you render the form with invalid data (after submitting for example).

You can overwrite the default label by passing it to the input method. You can also add a hint or even a placeholder. For boolean inputs, you can add an inline label as well:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username, label: 'Your username please' %>
  <%= f.input :password, hint: 'No special characters.' %>
  <%= f.input :email, placeholder: 'user@domain.com' %>
  <%= f.input :remember_me, inline_label: 'Yes, remember me' %>
  <%= f.button :submit %>
<% end %>
```

In some cases you may want to disable labels, hints or error. Or you may want to configure the html of any of them:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username, label_html: { class: 'my_class' } %>
  <%= f.input :password, hint: false, error_html: { id: 'password_error'} %>
  <%= f.input :password_confirmation, label: false %>
  <%= f.button :submit %>
<% end %>
```

It is also possible to pass any html attribute straight to the input, by using the :input_html option, for instance:

```
<%= simple_form_for @user do |f| %>
```

```
  <%= f.input :username, input_html: { class: 'special' } %>
  <%= f.input :password, input_html: { maxlength: 20 } %>
  <%= f.input :remember_me, input_html: { value: '1' } %>
  <%= f.button :submit %>
<% end %>
```

If you want to pass the same options to all inputs in the form (for example, a default class), you can use the :defaults option in simple_form_for. Specific options in input call will overwrite the defaults:

```
<%= simple_form_for @user, defaults: { input_html: { class: 'default_class' } } do |f| %>
  <%= f.input :username, input_html: { class: 'special' } %>
  <%= f.input :password, input_html: { maxlength: 20 } %>
  <%= f.input :remember_me, input_html: { value: '1' } %>
  <%= f.button :submit %>
<% end %>
```

Since **SimpleForm** generates a wrapper div around your label and input by default, you can pass any html attribute to that wrapper as well using the :wrapper_html option, like so:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username, wrapper_html: { class: 'username' } %>
  <%= f.input :password, wrapper_html: { id: 'password' } %>
  <%= f.input :remember_me, wrapper_html: { class: 'options' } %>
  <%= f.button :submit %>
<% end %>
```

Required fields are marked with an * prepended to their labels.

By default all inputs are required. When the form object has presence validations attached to its fields, **SimpleForm** tells required and optional fields apart. For performance reasons, this detection is skipped on validations that make use of conditional options, such as :if and :unless.

And of course, the required property of any input can be overwritten as needed:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :name, required: false %>
  <%= f.input :username %>
```

```
  <%= f.input :password %>
  <%= f.button :submit %>
<% end %>
```

SimpleForm also lets you overwrite the default input type it creates:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username %>
  <%= f.input :password %>
  <%= f.input :description, as: :text %>
  <%= f.input :accepts,     as: :radio_buttons %>
  <%= f.button :submit %>
<% end %>
```

So instead of a checkbox for the *accepts* attribute, you'll have a pair of radio buttons with yes/no labels and a text area instead of a text field for the description. You can also render boolean attributes using as: :select to show a dropdown.

It is also possible to give the :disabled option to **SimpleForm**, and it'll automatically mark the wrapper as disabled with a css class, so you can style labels, hints and other components inside the wrapper as well:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :username, disabled: true, hint: 'You cannot change your username.' %>
  <%= f.button :submit %>
<% end %>
```

SimpleForm accepts same options as their corresponding input type helper in Rails:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :date_of_birth, as: :date, start_year: Date.today.year - 90,
                              end_year: Date.today.year - 12, discard_day: true,
                              order: [:month, :year] %>
  <%= f.button :submit %>
<% end %>
```

SimpleForm also allows you to use label, hint, input_field, error and full_error helpers (please take a look at the rdocs for each method for more info):

```
<%= simple_form_for @user do |f| %>
  <%= f.label :username %>
  <%= f.input_field :username %>
  <%= f.hint 'No special characters, please!' %>
  <%= f.error :username, id: 'user_name_error' %>
  <%= f.full_error :token %>
  <%= f.submit 'Save' %>
<% end %>
```

Any extra option passed to these methods will be rendered as html option.

Collections

And what if you want to create a select containing the age from 18 to 60 in your form? You can do it overriding the :collection option:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :user %>
  <%= f.input :age, collection: 18..60 %>
  <%= f.button :submit %>
<% end %>
```

Collections can be arrays or ranges, and when a :collection is given the :select input will be rendered by default, so we don't need to pass the as: :select option. Other types of collection are :radio_buttons and :check_boxes. Those are added by **SimpleForm** to Rails set of form helpers (read Extra Helpers session below for more information).

Collection inputs accept two other options beside collections:

- *label_method* => the label method to be applied to the collection to retrieve the label (use this instead of the text_method option in collection_select)
- *value_method* => the value method to be applied to the collection to retrieve the value

Those methods are useful to manipulate the given collection. Both of these options also accept lambda/procs in case you want to calculate the value or label in a special way eg. custom translation. All other options given are sent straight to the underlying helper. For example, you can give prompt as:

```
f.input :age, collection: 18..60, prompt: "Select your age"
```

It is also possible to create grouped collection selects, that will use the html *optgroup* tags, like this:

```
f.input :country_id, collection: @continents, as: :grouped_select, group_method: :countries
```

Grouped collection inputs accept the same :label_method and :value_method options, which will be used to retrieve label/value attributes for the option tags. Besides that, you can give:

- *group_method* => the method to be called on the given collection to generate the options for each group (required)
- *group_label_method* => the label method to be applied on the given collection to retrieve the label for the *optgroup* (**SimpleForm** will attempt to guess the best one the same way it does with :label_method)

Priority

SimpleForm also supports :time_zone and :country. When using such helpers, you can give :priority as option to select which time zones and/or countries should be given higher priority:

```
f.input :residence_country, priority: [ "Brazil" ]
f.input :time_zone, priority: /US/
```

Those values can also be configured with a default value to be used site use through the SimpleForm.country_priority and SimpleForm.time_zone_priority helpers.

Note: While using country_select if you want to restrict to only a subset of countries for a specific drop down then you may use the :collection option:

```
f.input :shipping_country, priority: [ "Brazil" ], collection: [ "Australia", "Brazil", "New Zealand"]
```

Associations

To deal with associations, **SimpleForm** can generate select inputs, a series of radios buttons or check boxes. Lets see how it works: imagine you have a user model that belongs

to a company and has_and_belongs_to_many roles. The structure would be something like:

```
class User < ActiveRecord::Base
  belongs_to :company
  has_and_belongs_to_many :roles
end

class Company < ActiveRecord::Base
  has_many :users
end

class Role < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```

Now we have the user form:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :name %>
  <%= f.association :company %>
  <%= f.association :roles %>
  <%= f.button :submit %>
<% end %>
```

Simple enough, right? This is going to render a :select input for choosing the :company, and another :select input with :multiple option for the :roles. You can, of course, change it to use radio buttons and check boxes as well:

```
f.association :company, as: :radio_buttons
f.association :roles,   as: :check_boxes
```

The association helper just invokes input under the hood, so all options available to :select, :radio_buttons and :check_boxes are also available to association. Additionally, you can specify the collection by hand, all together with the prompt:

```
f.association :company, collection: Company.active.all(order: 'name'), prompt: "Choose a Company"
```

In case you want to declare different labels and values:

```
f.association :company, label_method: :company_name, value_method: :id, include_blank: false
```

Buttons

All web forms need buttons, right? **SimpleForm** wraps them in the DSL, acting like a proxy:

```
<%= simple_form_for @user do |f| %>
  <%= f.input :name %>
  <%= f.button :submit %>
<% end %>
```

The above will simply call submit. You choose to use it or not, it's just a question of taste.

Wrapping Rails Form Helpers

Say you wanted to use a rails form helper but still wrap it in **SimpleForm** goodness? You can, by calling input with a block like so:

```
<%= f.input :role do %>
  <%= f.select :role, Role.all.map { |r| [r.name, r.id, { class: r.company.id }] }, include_blank: true %>
<% end %>
```

In the above example, we're taking advantage of Rails 3's select method that allows us to pass in a hash of additional attributes for each option.

Extra helpers

SimpleForm also comes with some extra helpers you can use inside rails default forms without relying on `simple_form_for` helper. They are listed below.

Simple Fields For

Wrapper to use **SimpleForm** inside a default rails form. It works in the same way that the `field_for` Rails helper, but change the builder to use the `SimpleForm::FormBuilder`.

```
form_for @user do |f|
  f.simple_fields_for :posts do |posts_form|
    # Here you have all simple_form methods available
    posts_form.input :title
  end
end
```

Collection Radio Buttons

Creates a collection of radio inputs with labels associated (same API as collection_select):

```
form_for @user do |f|
  f.collection_radio_buttons :options, [[true, 'Yes'] ,[false, 'No']], :first, :last
end
```

```
<input id="user_options_true" name="user[options]" type="radio" value="true" />
<label class="collection_radio_buttons" for="user_options_true">Yes</label>
<input id="user_options_false" name="user[options]" type="radio" value="false" />
<label class="collection_radio_buttons" for="user_options_false">No</label>
```

Collection Check Boxes

Creates a collection of check boxes with labels associated (same API as collection_select):

```
form_for @user do |f|
  f.collection_check_boxes :options, [[true, 'Yes'] ,[false, 'No']], :first, :last
end
```

```
<input name="user[options][]" type="hidden" value="" />
<input id="user_options_true" name="user[options][]" type="checkbox" value="true" />
<label class="collection_check_box" for="user_options_true">Yes</label>
<input name="user[options][]" type="hidden" value="" />
<input id="user_options_false" name="user[options][]" type="checkbox" value="false" />
<label class="collection_check_box" for="user_options_false">No</label>
```

To use this with associations in your model, you can do the following:

```
form_for @user do |f|
  f.collection_check_boxes :role_ids, Role.all, :id, :name # using :roles here is not going to work.
end
```

Mappings/Inputs available

SimpleForm comes with a lot of default mappings:

Mapping	Input	Column Type
boolean	check box	boolean
string	text field	string
email	email field	string with name matching "email"
url	url field	string with name matching "url"
tel	tel field	string with name matching "phone"
password	password field	string with name matching "password"
search	search	-
text	text area	text
file	file field	string, responding to file methods
hidden	hidden field	-
integer	number field	integer
float	number field	float
decimal	number field	decimal
range	range field	-
datetime	datetime select	datetime/timestamp
date	date select	date
time	time select	time
select	collection select	belongs_to/has_many/has_and_belongs_to_many associations
radio_buttons	collection radio buttons	belongs_to
check_boxes	collection check boxes	has_many/has_and_belongs_to_many associations
country	country select	string with name matching "country"
time_zone	time zone select	string with name matching "time_zone"

Custom inputs

It is very easy to add custom inputs to **SimpleForm**. For instance, if you want to add a custom input that extends the string one, you just need to add this file:

```
# app/inputs/currency_input.rb
class CurrencyInput < SimpleForm::Inputs::Base
  def input
    "$ #{@builder.text_field(attribute_name, input_html_options)}".html_safe
  end
end
```

And use it in your views:

```
f.input :money, as: :currency
```

You can also redefine existing **SimpleForm** inputs by creating a new class with the same name. For instance, if you want to wrap date/time/datetime in a div, you can do:

```
# app/inputs/date_time_input.rb
class DateTimeInput < SimpleForm::Inputs::DateTimeInput
  def input
    template.content_tag(:div, super)
  end
end
```

Or if you want to add a class to all the select fields you can do:

```
# app/inputs/collection_select_input.rb
class CollectionSelectInput < SimpleForm::Inputs::CollectionSelectInput
  def input_html_classes
    super.push('chosen')
  end
end
```

Custom form builder

You can create a custom form builder that uses **SimpleForm**.

Create a helper method that calls simple_form_for with a custom builder:

```
def custom_form_for(object, *args, &block)
  options = args.extract_options!
```

```
  simple_form_for(object, *(args << options.merge(builder: CustomFormBuilder)), &block)
end
```

Create a form builder class that inherits from SimpleForm::FormBuilder.

```
class CustomFormBuilder < SimpleForm::FormBuilder
  def input(attribute_name, options = {}, &block)
    options[:input_html].merge! class: 'custom'
    super
  end
end
```

SimpleForm uses all power of I18n API to lookup labels, hints and placeholders. To customize your forms you can create a locale file like this:

```
en:
  simple_form:
    labels:
      user:
        username: 'User name'
        password: 'Password'
    hints:
      user:
        username: 'User name to sign in.'
        password: 'No special characters, please.'
    placeholders:
      user:
        username: 'Your username'
        password: '****'
```

And your forms will use this information to render the components for you.

SimpleForm also lets you be more specific, separating lookups through actions for labels, hints and placeholders. Let's say you want a different label for new and edit actions, the locale file would be something like:

```
en:
  simple_form:
    labels:
```

```
    user:
      username: 'User name'
      password: 'Password'
      edit:
        username: 'Change user name'
        password: 'Change password'
```

This way **SimpleForm** will figure out the right translation for you, based on the action being rendered. And to be a little bit DRYer with your locale file, you can specify defaults for all models under the 'defaults' key:

```
en:
  simple_form:
    labels:
      defaults:
        username: 'User name'
        password: 'Password'
        new:
          username: 'Choose a user name'
    hints:
      defaults:
        username: 'User name to sign in.'
        password: 'No special characters, please.'
    placeholders:
      defaults:
        username: 'Your username'
        password: '****'
```

SimpleForm will always look for a default attribute translation under the "defaults" key if no specific is found inside the model key. Note that this syntax is different from 1.x. To migrate to the new syntax, just move "labels.#{attribute}" to "labels.defaults.#{attribute}".

In addition, **SimpleForm** will fallback to default human_attribute_name from Rails when no other translation is found for labels. Finally, you can also overwrite any label, hint or placeholder inside your view, just by passing the option manually. This way the I18n lookup will be skipped.

SimpleForm also has support for translating options in collection helpers. For instance, given a User with a :gender attribute, you might want to create a select box showing

translated labels that would post either male or female as value. With **SimpleForm** you could create an input like this:

```
f.input :gender, collection: [:male, :female]
```

And **SimpleForm** will try a lookup like this in your locale file, to find the right labels to show:

```
en:
  simple_form:
    options:
      user:
        gender:
          male: 'Male'
          female: 'Female'
```

You can also use the defaults key as you would do with labels, hints and placeholders. It is important to notice that **SimpleForm** will only do the lookup for options if you give a collection composed of symbols only. This is to avoid constant lookups to I18n.

It's also possible to translate buttons, using Rails' built-in I18n support:

```
en:
  helpers:
    submit:
      user:
        create: "Add %{model}"
        update: "Save Changes"
```

There are other options that can be configured through I18n API, such as required text and boolean. Be sure to check our locale file or the one copied to your application after you run rails generate simple_form:install.

It should be noted that translations for labels, hints and placeholders for a namespaced model, e.g. Admin::User, should be placed under admin_user, not under admin/user. This is different from how translations for namespaced model and attribute names are defined:

```
en:
  activerecord:
```

```
  models:
    admin/user: User
  attributes:
    admin/user:
      name: Name
```

They should be placed under admin/user. Form labels, hints and placeholders for those attributes, though, should be placed under admin_user:

```
en:
  simple_form:
    labels:
      admin_user:
        name: Name
```

This difference exists because **SimpleForm** relies on object_name provided by Rails' FormBuilder to determine the translation path for a given object instead of i18n_key from the object itself. Thus, similarly, if a form for an Admin::User object is defined by calling simple_form_for @admin_user, as: :some_user, **SimpleForm** will look for translations under some_user instead of admin_user.

Configuration

SimpleForm has several configuration options. You can read and change them in the initializer created by **SimpleForm**, so if you haven't executed the command below yet, please do:

```
rails generate simple_form:install
```

The wrappers API

With **SimpleForm** you can configure how your components will be rendered using the wrappers API. The syntax looks like this:

```
config.wrappers tag: :div, class: :input,
                error_class: :field_with_errors do |b|
```

```
  # Form extensions
  b.use :html5
  b.optional :pattern
  b.use :maxlength
  b.use :placeholder
  b.use :readonly

  # Form components
  b.use :label_input
  b.use :hint,  wrap_with: { tag: :span, class: :hint }
  b.use :error, wrap_with: { tag: :span, class: :error }
end
```

The *Form components* will generate the form tags like labels, inputs, hints or errors contents. The available components are:

```
:label          # The <label> tag alone
:input          # The <input> tag alone
:label_input    # The <label> and the <input> tags
:hint           # The hint for the input
:error          # The error for the input
```

The *Form extensions* are used to generate some attributes or perform some lookups on the model to add extra information to your components.

You can create new *Form components* using the wrappers API as in the following example:

```
config.wrappers do |b|
  b.use :placeholder
  b.use :label_input
  b.wrapper tag: :div, class: 'separator' do |component|
    component.use :hint,  wrap_with: { tag: :span, class: :hint }
    component.use :error, wrap_with: { tag: :span, class: :error }
  end
end
```

this will wrap the hint and error components within a `div` tag using the class `'separator'`.

If you want to customize the custom *Form components* on demand you can give it a name like this:

```
config.wrappers do |b|
  b.use :placeholder
  b.use :label_input
  b.wrapper :my_wrapper, tag: :div, class: 'separator' do |component|
    component.use :hint,  wrap_with: { tag: :span, class: :hint }
    component.use :error, wrap_with: { tag: :span, class: :error }
  end
end
```

and now you can pass options to your input calls to customize the :my_wrapper *Form component*.

```
# Completely turns off the custom wrapper
f.input :name, my_wrapper: false

# Configure the html
f.input :name, my_wrapper_html: { id: 'special_id' }

# Configure the tag
f.input :name, my_wrapper_tag: :p
```

You can also define more than one wrapper and pick one to render in a specific form or input. To define another wrapper you have to give it a name, as the follow:

```
config.wrappers :small do |b|
  b.use :placeholder
  b.use :label_input
end
```

and use it in this way:

```
# Specifying to whole form
simple_form_for @user, wrapper: :small do |f|
  f.input :name
end
```

```
# Specifying to one input
simple_form_for @user do |f|
  f.input :name, wrapper: :small
end
```

SimpleForm also allows you to use optional elements. For instance, let's suppose you want to use hints or placeholders, but you don't want them to be generated automatically. You can set their default values to `false` or use the `optional` method. Is preferible to use the `optional` syntax:

```
config.wrappers placeholder: false do |b|
  b.use :placeholder
  b.use :label_input
  b.wrapper tag: :div, class: 'separator' do |component|
    component.optional :hint, wrap_with: { tag: :span, class: :hint }
    component.use :error, wrap_with: { tag: :span, class: :error }
  end
end
```

By setting it as `optional`, a hint will only be generated when `hint: true` is explicitly used. The same for placehold.

HTML 5 Notice

By default, **SimpleForm** will generate input field types and attributes that are supported in HTML5, but are considered invalid HTML for older document types such as HTML4 or XHTML1.0. The HTML5 extensions include the new field types such as email, number, search, url, tel, and the new attributes such as required, autofocus, maxlength, min, max, step.

Most browsers will not care, but some of the newer ones - in particular Chrome 10+ - use the required attribute to force a value into an input and will prevent form submission without it. Depending on the design of the application this may or may not be desired. In many cases it can break existing UI's.

It is possible to disable all HTML 5 extensions in **SimpleForm** with the following configuration:

```
SimpleForm.html5 = false # default is true
```

If you want to have all other HTML 5 features, such as the new field types, you can disable only the browser validation:

```
SimpleForm.browser_validations = false # default is true
```

This option adds a new `novalidate` property to the form, instructing it to skip all HTML 5 validation. The inputs will still be generated with the required and other attributes, that might help you to use some generic javascript validation.

You can also add `novalidate` to a specific form by setting the option on the form itself:

```
<%= simple_form_for(resource, html: {novalidate: true}) do |form| %>
```

Please notice that any of the configurations above will disable the `placeholder` component, which is an HTML 5 feature. We believe most of the newest browsers are handling this attribute fine, and if they aren't, any plugin you use would take of using the placeholder attribute to do it. However, you can disable it if you want, by removing the placeholder component from the components list in **SimpleForm** configuration file.

Information

Google Group

If you have any questions, comments, or concerns please use the Google Group instead of the GitHub Issues tracker:

http://groups.google.com/group/plataformatec-simpleform [10]

RDocs

You can view the **SimpleForm** documentation in RDoc format here:

http://rubydoc.info/github/plataformatec/simple_form/master/frames [11]

If you need to use **SimpleForm** with Rails 2.3, you can always run `gem server` from the command line after you install the gem to access the old documentation.

Bug reports

If you discover any bugs, feel free to create an issue on GitHub. Please add as much information as possible to help us fixing the possible bug. We also encourage you to help even more by forking and sending us a pull request.

https://github.com/plataformatec/simple_form/issues [12]

Maintainers

License

1. http://simple-form.plataformatec.com.br/
2. https://github.com/plataformatec/simple_form/tree/v1.5
3. https://rubygems.org/gems/country_select
4. http://twitter.github.com/bootstrap
5. http://twitter.github.com/bootstrap
6. https://github.com/rafaelfranca/simple_form-bootstrap
7. http://simple-form-bootstrap.plataformatec.com.br/
8. http://foundation.zurb.com/
9. http://foundation.zurb.com/docs/rails.php
10. http://groups.google.com/group/plataformatec-simpleform
11. http://rubydoc.info/github/plataformatec/simple_form/master/frames
12. https://github.com/plataformatec/simple_form/issues

13. http://plataformatec.com.br/