Installation

Active Admin is a Ruby Gem.

```
gem 'activeadmin'

# Plus integrations with:
gem 'devise'
gem 'cancan' # or cancancan
gem 'draper'
gem 'pundit'
```

More accurately, it's a Rails Engine that can be injected into your existing Ruby on Rails application.

Setting up Active Admin

After installing the gem, you need to run the generator. Here are your options:

- If you don't want to use Devise, run it with `--skip-users`:

  ```
  rails g active_admin:install --skip-users
  ```

- If you want to use an existing user class, provide it as an argument:

  ```
  rails g active_admin:install User
  ```

- Otherwise, with no arguments we will create an `AdminUser` class to use with Devise:

  ```
  rails g active_admin:install
  ```

The generator adds these core files, among others:

```
app/admin/dashboard.rb
app/assets/javascripts/active_admin.js.coffee
app/assets/stylesheets/active_admin.scss
config/initializers/active_admin.rb
```

Now, migrate and seed your database before starting the server:

```
rake db:migrate
rake db:seed
rails server
```

Visit `http://localhost:3000/admin` and log in as the default user:

- **User**: admin@example.com
- **Password**: password

Voila! You're on your brand new Active Admin dashboard.

To register an existing model with Active Admin:

```
rails generate active_admin:resource MyModel
```

This creates a file at `app/admin/my_model.rb` to set up the UI; refresh your browser to see it.

Upgrading

When upgrading to a new version, it's a good idea to check the CHANGELOG.

To update the JS & CSS assets:

```
rails generate active_admin:assets
```

You should also sync these files with their counterparts in the AA source code:

- app/admin/dashboard.rb ~>
- config/initializers/active_admin.rb ~>

Gem compatibility

will_paginate

If you use `will_paginate` in your app, you need to configure an initializer for Kaminari to avoid conflicts.

```
# config/initializers/kaminari.rb
Kaminari.configure do |config|
  config.page_method_name = :per_page_kaminari
end
```

If you are also using Draper, you may want to make sure `per_page_kaminari` is delegated correctly:

```
Draper::CollectionDecorator.send :delegate, :per_page_kaminari
```

simple_form

If you're getting the error `wrong number of arguments (6 for 4..5)`, read #2703.

General Configuration

You can configure Active Admin settings in `config/initializers/active_admin.rb`. Here are a few common configurations:

Authentication

Active Admin requires two settings to authenticate and use the current user within your application.

- the method controllers used to force authentication

```
config.authentication_method = :authenticate_admin_user!
```

- the method used to access the current user

```
config.current_user_method = :current_admin_user
```

Both of these settings can be set to false to turn off authentication.

```
config.authentication_method = false
config.current_user_method   = false
```

Site Title Options

Every page has what's called the site title on the left side of the menu bar. If you want, you can customize it.

```
config.site_title        = "My Admin Site"
config.site_title_link   = "/"
config.site_title_image  = "site_image.png"
config.site_title_image  = "http://www.google.com/images/logos/google_logo_41.png"
config.site_title_image  = ->(context) { context.current_user.company.logo_url }
```

Internationalization (I18n)

To translate Active Admin to a new language or customize an existing translation, you can copy config/locales/en.yml to your application's `config/locales` folder and update it. We welcome new/updated translations, so feel free to contribute! To translate third party gems like devise, use for example devise-i18n.

Localize Format For Dates and Times

Active Admin sets `:long` as default localize format for dates and times. If you want, you can customize it.

```
config.localize_format = :short
```

Namespaces

When registering resources in Active Admin, they are loaded into a namespace. The default namespace is "admin".

```
# app/admin/posts.rb
ActiveAdmin.register Post do
  # ...
end
```

The Post resource will be loaded into the "admin" namespace and will be available at `/admin/posts`.

Each namespace holds on to its own settings that inherit from the application's configuration.

For example, if you have two namespaces (`:admin` and `:super_admin`) and want to have different site title's for each, you can use the `config.namespace(name)` block within the initializer file to configure them individually.

```
ActiveAdmin.setup do |config|
  config.site_title = "My Default Site Title"

  config.namespace :admin do |admin|
    admin.site_title = "Admin Site"
  end

  config.namespace :super_admin do |super_admin|
    super_admin.site_title = "Super Admin Site"
  end
end
```

Each setting available in the Active Admin setup block is configurable on a per namespace basis.

Load paths

By default Active Admin files go inside `app/admin/`. You can change this directory in the initializer file:

```
ActiveAdmin.setup do |config|
  config.load_paths = [File.join(Rails.root, "app", "ui")]
end
```

Comments

By default Active Admin includes comments on resources. Sometimes, this is undesired. To disable comments:

```
# For the entire application:
ActiveAdmin.setup do |config|
  config.comments = false
end

# For a namespace:
ActiveAdmin.setup do |config|
  config.namespace :admin do |admin|
    admin.comments = false
  end
end

# For a given resource:
ActiveAdmin.register Post do
  config.comments = false
end
```

You can change the name under which comments are registered:

```
config.comments_registration_name = 'AdminComment'
```

You can change the order for the comments and you can change the column to be used for ordering:

```
config.comments_order = 'created_at ASC'
```

You can disable the menu item for the comments index page:

```
config.comments_menu = false
```

You can customize the comment menu:

```
config.comments_menu = { parent: 'Admin', priority: 1 }
```

Utility Navigation

The “utility navigation” shown at the top right normally shows the current user and a link to log out. However, the utility navigation is just like any other menu in the system; you can provide your own menu to be rendered in its place.

```
ActiveAdmin.setup do |config|
  config.namespace :admin do |admin|
    admin.build_menu :utility_navigation do |menu|
      menu.add label: "ActiveAdmin.info", url: "http://www.activeadmin.info",
                                          html_options: { target: :blank }
      admin.add_current_user_to_menu  menu
      admin.add_logout_button_to_menu menu
    end
  end
end
```

Footer Customization

By default, Active Admin displays a “Powered by ActiveAdmin” message on every page. You can override this message and show domain-specific messaging:

```
config.footer = "MyApp Revision v1.3"
```

Working with Resources

Every Active Admin resource corresponds to a Rails model. So before creating a resource you must first create a Rails model for it.

Create a Resource

The basic command for creating a resource is `rails g active_admin:resource Post`. The generator will produce an empty `app/admin/post.rb` file like so:

```
ActiveAdmin.register Post do
  # everything happens here :D
end
```

Setting up Strong Parameters

Use the `permit_params` method to define which attributes may be changed:

```
ActiveAdmin.register Post do
  permit_params :title, :content, :publisher_id
end
```

Any form field that sends multiple values (such as a HABTM association, or an array attribute) needs to pass an empty array to `permit_params`:

If your HABTM is `roles`, you should permit `role_ids: []`

```
ActiveAdmin.register Post do
  permit_params :title, :content, :publisher_id, role_ids: []
end
```

Nested associations in the same form also require an array, but it needs to be filled with any attributes used.

```
ActiveAdmin.register Post do
  permit_params :title, :content, :publisher_id,
    tags_attributes: [:id, :name, :description, :_destroy]
end

# Note that `accepts_nested_attributes_for` is still required:
class Post < ActiveRecord::Base
  accepts_nested_attributes_for :tags, allow_destroy: true
end
```

If you want to dynamically choose which attributes can be set, pass a block:

```
ActiveAdmin.register Post do
  permit_params do
    params = [:title, :content, :publisher_id]
    params.push :author_id if current_user.admin?
    params
  end
end
```

If your resource is nested, declare `permit_params` after `belongs_to`:

```
ActiveAdmin.register Post do
  belongs_to :user
  permit_params :title, :content, :publisher_id
end
```

The `permit_params` call creates a method called `permitted_params`. You should use this method when overriding `create` or `update` actions:

```
ActiveAdmin.register Post do
  controller do
    def create
      # Good
      @post = Post.new(permitted_params[:post])
      # Bad
      @post = Post.new(params[:post])

      if @post.save
        # ...
      end
    end
  end
end
```

Disabling Actions on a Resource

All CRUD actions are enabled by default. These can be disabled for a given resource:

```
ActiveAdmin.register Post do
  actions :all, except: [:update, :destroy]
end
```

Renaming Action Items

You can use translations to override labels and page titles for actions such as new, edit, and destroy by providing a resource specific translation. For example, to change ‘New Offer’ to ‘Make an Offer’ add the following in config/locales/[en].yml:

```
en:
  active_admin:
    resources:
      offer:
        new_model: 'Make an Offer'
```

Rename the Resource

By default, any references to the resource (menu, routes, buttons, etc) in the interface will use the name of the class. You can rename the resource by using the `:as` option.

```
ActiveAdmin.register Post, as: "Article"
```

The resource will then be available at `/admin/articles`.

Customize the Namespace

We use the `admin` namespace by default, but you can use anything:

```
# Available at /today/posts
ActiveAdmin.register Post, namespace: :today

# Available at /posts
ActiveAdmin.register Post, namespace: false
```

Customize the Menu

The resource will be displayed in the global navigation by default. To disable the resource from being displayed in the global navigation:

```
ActiveAdmin.register Post do
  menu false
end
```

The menu method accepts a hash with the following options:

- `:label` - The string or proc label to display in the menu. If it's a proc, it will be called each time the menu is rendered.
- `:parent` - The string id (or label) of the parent used for this menu
- `:if` - A block or a symbol of a method to call to decide if the menu item should be displayed
- `:priority` - The integer value of the priority, which defaults to `10`

Labels

To change the name of the label in the menu:

```
ActiveAdmin.register Post do
  menu label: "My Posts"
end
```

If you want something more dynamic, pass a proc instead:

```
ActiveAdmin.register Post do
  menu label: proc{ I18n.t "mypost" }
end
```

Menu Priority

Menu items are sorted first by their numeric priority, then alphabetically. Since every menu by default has a priority of `10`, the menu is normally alphabetical.

You can easily customize this:

```
ActiveAdmin.register Post do
  menu priority: 1 # so it's on the very left
end
```

Conditionally Showing / Hiding Menu Items

Menu items can be shown or hidden at runtime using the `:if` option.

```
ActiveAdmin.register Post do
  menu if: proc{ current_user.can_edit_posts? }
end
```

The proc will be called in the context of the view, so you have access to all your helpers and current user session information.

Drop Down Menus

In many cases, a single level navigation will not be enough to manage a large application. In that case, you can group your menu items under a parent menu item.

```
ActiveAdmin.register Post do
  menu parent: "Blog"
end
```

Note that the "Blog" parent menu item doesn't even have to exist yet; it can be dynamically generated for you.

Customizing Parent Menu Items

All of the options given to a standard menu item are also available to parent menu items. In the case of complex parent menu items, you should configure them in the Active Admin initializer.

```
# config/initializers/active_admin.rb
config.namespace :admin do |admin|
  admin.build_menu do |menu|
    menu.add label: 'Blog', priority: 0
  end
end

# app/admin/post.rb
ActiveAdmin.register Post do
  menu parent: 'Blog'
end
```

Dynamic Parent Menu Items

While the above works fine, what if you want a parent menu item with a dynamic name? Well, you have to refer to it by its `:id`.

```
# config/initializers/active_admin.rb
config.namespace :admin do |admin|
  admin.build_menu do |menu|
    menu.add id: 'blog', label: proc{"Something dynamic"}, priority: 0
  end
end

# app/admin/post.rb
ActiveAdmin.register Post do
  menu parent: 'blog'
end
```

Adding Custom Menu Items

Sometimes it's not enough to just customize the menu label. In this case, you can customize the menu for the namespace within the Active Admin initializer.

```
# config/initializers/active_admin.rb
config.namespace :admin do |admin|
  admin.build_menu do |menu|
    menu.add label: "The Application", url: "/", priority: 0

    menu.add label: "Sites" do |sites|
      sites.add label: "Google",
                url: "http://google.com",
                html_options: { target: :blank }

      sites.add label: "Facebook",
                url: "http://facebook.com"

      sites.add label: "Github",
                url: "http://github.com"
    end
  end
end
```

This will be registered on application start before your resources are loaded.

Scoping the queries

If your administrators have different access levels, you may sometimes want to scope what they have access to. Assuming your User model has the proper has_many relationships, you can simply scope the listings and finders like so:

```
ActiveAdmin.register Post do
  scope_to :current_user # limits the accessible posts to `current_user.posts`

  # Or if the association doesn't have the default name:
  scope_to :current_user, association_method: :blog_posts

  # Finally, you can pass a block to be called:
  scope_to do
```

```
    User.most_popular_posts
  end
end
```

You can also conditionally apply the scope:

```
ActiveAdmin.register Post do
  scope_to :current_user, if:     proc{ current_user.limited_access? }
  scope_to :current_user, unless: proc{ current_user.admin? }
end
```

Eager loading

A common way to increase page performance is to elimate N+1 queries by eager loading associations:

```
ActiveAdmin.register Post do
  includes :author, :categories
end
```

Customizing resource retrieval

Our controllers are built on Inherited Resources, so you can use all of its features.

If you need to customize the collection properties, you can overwrite the `scoped_collection` method.

```
ActiveAdmin.register Post do
  controller do
    def scoped_collection
      end_of_association_chain.where(visibility: true)
    end
  end
end
```

If you need to completely replace the record retrieving code (e.g., you have a custom `to_param` implementation in your models), override the `resource` method on the controller:

```
ActiveAdmin.register Post do
  controller do
    def find_resource
      scoped_collection.where(id: params[:id]).first!
    end
  end
end
```

Note that if you use an authorization library like CanCan, you should be careful to not write code like this, otherwise **your authorization rules won't be applied**:

```
ActiveAdmin.register Post do
  controller do
    def find_resource
      Post.where(id: params[:id]).first!
    end
  end
end
```

Belongs To

It's common to want to scope a series of resources to a relationship. For example a Project may have many Milestones and Tickets. To nest the resource within another, you can use the `belongs_to` method:

```
ActiveAdmin.register Project
ActiveAdmin.register Ticket do
  belongs_to :project
end
```

Projects will be available as usual and tickets will be available by visiting `/admin/projects/1/tickets` assuming that a Project with the id of 1 exists. Active Admin does not add “Tickets” to the global navigation because the routes can only be generated when there is a project id.

To create links to the resource, you can add them to a sidebar (one of the many possibilities for how you may with to handle your user interface):

```
ActiveAdmin.register Project do

  sidebar "Project Details", only: [:show, :edit] do
    ul do
      li link_to "Tickets",    admin_project_tickets_path(resource)
      li link_to "Milestones", admin_project_milestones_path(resource)
    end
  end
end

ActiveAdmin.register Ticket do
  belongs_to :project
end

ActiveAdmin.register Milestone do
  belongs_to :project
end
```

In some cases (like Projects), there are many sub resources and you would actually like the global navigation to switch when the user navigates “into” a project. To accomplish this, Active Admin stores the `belongs_to` resources in a separate menu which you can use if you so wish. To use:

```
ActiveAdmin.register Ticket do
  belongs_to :project
  navigation_menu :project
end

ActiveAdmin.register Milestone do
  belongs_to :project
  navigation_menu :project
end
```

Now, when you navigate to the tickets section, the global navigation will only display “Tickets” and “Milestones”. When you navigate back to a non-belongs_to resource, it will switch back to the default menu.

You can also defer the menu lookup until runtime so that you can dynamically show different menus, say perhaps based on user permissions. For example:

```
ActiveAdmin.register Ticket do
  belongs_to :project
  navigation_menu do
    authorized?(:manage, SomeResource) ? :project : :restricted_menu
  end
end
```

If you still want your `belongs_to` resources to be available in the default menu and through non-nested routes, you can use the `:optional` option. For example:

```
ActiveAdmin.register Ticket do
  belongs_to :project, optional: true
end
```

Customizing the Index Page

Filtering and listing resources is one of the most important tasks for administering a web application. Active Admin provides many different tools for you to build a compelling interface into your data for the admin staff.

Built in, Active Admin has the following index renderers:

- *Table*: A table drawn with each row being a resource (View Table Docs)
- *Grid*: A set of rows and columns each cell being a resource (View Grid Docs)
- *Blocks*: A set of rows (not tabular) each row being a resource (View Blocks Docs)
- *Blog*: A title and body content, similar to a blog index (View Blog Docs)

All index pages also support scopes, filters, pagination, action items, and sidebar sections.

Multiple Index Pages

Sometime you may want more than one index page for a resource to represent different views to the user. If multiple index pages exist, Active Admin will automatically build links at the top of the default index page. Including multiple views is simple and requires creating multiple index components in your resource.

```
index do
  id_column
  column :image_title
  actions
end

index as: :grid do |product|
  link_to image_tag(product.image_path), admin_product_path(product)
end
```

The first index component will be the default index page unless you indicate otherwise by setting `:default` to true.

```
index do
  column :image_title
  actions
end

index as: :grid, default: true do |product|
  link_to image_tag(product.image_path), admin_product_path(product)
end
```

Custom Index

Active Admin does not limit the index page to be a table, block, blog or grid. If you've created your own custom index page it can be included by setting `:as` to the class of the index component you created.

```
index as: ActiveAdmin::Views::IndexAsMyIdea do
  column :image_title
  actions
end
```

Index Filters

By default the index screen includes a "Filters" sidebar on the right hand side with a filter for each attribute of the registered model. You can customize the filters that are displayed as well as the type of widgets they use.

To display a filter for an attribute, use the `filter` method

```
ActiveAdmin.register Post do
  filter :title
end
```

Out of the box, Active Admin supports the following filter types:

- *:string* - A search field
- *:date_range* - A start and end date field with calendar inputs
- *:numeric* - A drop down for selecting “Equal To”, “Greater Than” or “Less Than” and an input for a value.
- *:select* - A drop down which filters based on a selected item in a collection or all.
- *:check_boxes* - A list of check boxes users can turn on and off to filter

By default, Active Admin will pick the most relevant filter based on the attribute type. You can force the type by passing the `:as` option.

```
filter :author, as: :check_boxes
```

The `:check_boxes` and `:select` types accept options for the collection. By default it attempts to create a collection based on an association. But you can pass in the collection as a proc to be called at render time.

```
filter :author, as: :check_boxes, collection: proc { Author.all }
```

To override options for string or numeric filter pass `filters` option.

```
  filter :title, filters: [:starts_with, :ends_with]
```

Also, if you don’t need the select with the options ‘contains’, ‘equals’, ‘starts_with’ or ‘ends_with’ just add the option to the filter name with an underscore.

For example:

```
filter :name_equals
# or
filter :name_contains
```

You can change the filter label by passing a label option:

```
filter :author, label: 'Something else'
```

By default, Active Admin will try to use ActiveModel I18n to determine the label.

You can also filter on more than one attribute of a model using the Ransack search predicate syntax. If using a custom search method, you will also need to specify the field type using `:as` and the label.

```
filter :first_name_or_last_name_cont, as: :string, label: "Name"
```

Filters can also be disabled for a resource, a namespace or the entire application.

To disable for a specific resource:

```
ActiveAdmin.register Post do
  config.filters = false
end
```

To disable for a namespace, in the initializer:

```
ActiveAdmin.setup do |config|
  config.namespace :my_namespace do |my_namespace|
    my_namespace.filters = false
  end
end
```

Or to disable for the entire application:

```
ActiveAdmin.setup do |config|
  config.filters = false
end
```

You can also add a filter and still preserve the default filters:

```
preserve_default_filters!
filter :author
```

Or you can also remove a filter and still preserve the default filters:

```
preserve_default_filters!
remove_filter :id
```

Index Scopes

You can define custom scopes for your index page. This will add a tab bar above the index table to quickly filter your collection on pre-defined scopes. There are a number of ways to define your scopes:

```
scope :all, default: true

# assumes the model has a scope called ':active'
scope :active

# renames model scope ':leaves' to ':subcategories'
scope "Subcategories", :leaves

# Dynamic scope name
scope ->{ Date.today.strftime '%A' }, :published_today

# custom scope not defined on the model
scope("Inactive") { |scope| scope.where(active: false) }

# conditionally show a custom controller scope
scope "Published", if: -> { current_admin_user.can? :manage, Posts } do |posts|
  posts.published
end
```

Scopes can be labelled with a translation, e.g. `activerecord.scopes.invoice.expired`.

Index default sort order

You can define the default sort order for index pages:

```
ActiveAdmin.register Post do
  config.sort_order = 'name_asc'
end
```

Index pagination

You can set the number of records per page as default:

```
ActiveAdmin.setup do |config|
  config.default_per_page = 30
end
```

You can set the number of records per page per resources:

```
ActiveAdmin.register Post do
  config.per_page = 10
end
```

You can change it per request / action too:

```
controller do
  before_action only: :index do
    @per_page = 100
  end
end
```

You can also disable pagination:

```
ActiveAdmin.register Post do
  config.paginate = false
end
```

If you have a very large database, you might want to disable `SELECT COUNT(*)` queries caused by the pagination info at the bottom of the page:

```
ActiveAdmin.register Post do
  index pagination_total: false do
    # ...
  end
end
```

Customizing Download Links

You can easily remove or customize the download links you want displayed:

```
# Per resource:
ActiveAdmin.register Post do

  index download_links: false
  index download_links: [:pdf]
  index download_links: proc{ current_user.can_view_download_links? }

end

# For the entire application:
ActiveAdmin.setup do |config|

  config.download_links = false
  config.download_links = [:csv, :xml, :json, :pdf]
  config.download_links = proc { current_user.can_view_download_links? }

end
```

Note: you have to actually implement PDF rendering for your action, ActiveAdmin does not provide this feature. This setting just allows you to specify formats that you want to show up under the index collection.

You'll need to use a PDF rendering library like PDFKit or WickedPDF to get the PDF generation you want.

Customizing the CSV format

Active Admin provides CSV file downloads on the index screen for each Resource. By default it will render a CSV file with all the content columns of your registered model.

Customizing the CSV format is as simple as customizing the index page.

```
ActiveAdmin.register Post do
  csv do
    column :title
    column(:author) { |post| post.author.full_name }
    column('bODY', humanize_name: false) # preserve case
  end
end
```

You can also set custom CSV settings for an individual resource:

```
ActiveAdmin.register Post do
  csv force_quotes: true, col_sep: ';', column_names: false do
    column :title
    column(:author) { |post| post.author.full_name }
  end
end
```

Or system-wide:

```
# config/initializers/active_admin.rb

# Set the CSV builder separator
config.csv_options = { col_sep: ';' }

# Force the use of quotes
config.csv_options = { force_quotes: true }
```

You can customize the filename by overriding `csv_filename` in the controller block.

```
ActiveAdmin.register User do
  controller do
    def csv_filename
      'User Details.csv'
    end
  end
end
```

Streaming

By default Active Admin streams the CSV response to your browser as it's generated. This is good because it prevents request timeouts, for example the infamous H12 error on Heroku.

However if an exception occurs while generating the CSV, the request will eventually time out, with the last line containing the exception message. CSV streaming is disabled in development to help debug these exceptions. That lets you use tools like better_errors and web-console to debug the issue. If you want to customize the environments where CSV streaming is disabled, you can change this setting:

```
# config/initializers/active_admin.rb

config.disable_streaming_in = ['development', 'staging']
```

Forms

Active Admin gives you complete control over the output of the form by creating a thin DSL on top of Formtastic:

```
ActiveAdmin.register Post do

  form title: 'A custom title' do |f|
    inputs 'Details' do
      input :title
      input :published_at, label: "Publish Post At"
      li "Created at #{f.object.created_at}" unless f.object.new_record?
      input :category
    end
    panel 'Markup' do
      "The following can be used in the content below..."
    end
    inputs 'Content', :body
    para "Press cancel to return to the list without saving."
    actions
  end

end
```

For more details, please see Formtastic's documentation.

Default

Resources come with a default form defined as such:

```
form do |f|
  f.semantic_errors # shows errors on :base
  f.inputs          # builds an input field for every attribute
  f.actions         # adds the 'Submit' and 'Cancel' buttons
end
```

Partials

If you want to split a custom form into a separate partial use:

```
ActiveAdmin.register Post do
  form partial: 'form'
end
```

Which looks for something like this:

```
# app/views/admin/posts/_form.html.arb
insert_tag active_admin_form_for resource do |f|
  inputs :title, :body
  actions
end
```

This is a regular Rails partial so any template engine may be used.

You can also use the `ActiveAdmin::FormBuilder` as builder in your Formtastic Form for use the same helpers are used in the admin file:

```
  = semantic_form_for [:admin, @post], builder: ActiveAdmin::FormBuilder do |f|
    = f.inputs "Details" do
      = f.input :title
    - f.has_many :taggings, sortable: :position, sortable_start: 1 do |t|
      - t.input :tag
    = f.actions
```

Nested Resources

You can create forms with nested models using the `has_many` method, even if your model uses `has_one`:

```
ActiveAdmin.register Post do

  form do |f|
    f.inputs 'Details' do
      f.input :title
      f.input :published_at, label: 'Publish Post At'
    end
    f.inputs 'Content', :body
    f.inputs do
      f.has_many :categories, heading: 'Themes',
                              allow_destroy: true,
                              new_record: false do |a|
        a.input :title
      end
    end
    f.inputs do
      f.has_many :taggings, sortable: :position, sortable_start: 1 do |t|
        t.input :tag
      end
    end
    f.inputs do
      f.has_many :comment,
                 new_record: 'Leave Comment',
                 allow_destroy: -> { |c| c.author?(current_admin_user) } do |b|
        b.input :body
      end
    end
    f.actions
  end

end
```

The `:allow_destroy` option adds a checkbox to the end of the nested form allowing removal of the child object upon submission. Be sure to set `allow_destroy: true` on the association to use this option. It is possible to associate `:allow_destroy` with a string or a symbol, corresponding to the name of a child object's method that will get called, or with a Proc object. The Proc object receives the child object as a parameter and should return either true or false.

The `:heading` option adds a custom heading. You can hide it entirely by passing `false`.

The `:new_record` option controls the visibility of the new record button (shown by default). If you pass a string, it will be used as the text for the new record button.

The `:sortable` option adds a hidden field and will enable drag & drop sorting of the children. It expects the name of the column that will store the index of each child.

The `:sortable_start` option sets the value (0 by default) of the first position in the list.

Datepicker

ActiveAdmin offers the `datepicker` input, which uses the jQuery UI datepicker. The datepicker input accepts any of the options available to the standard jQueryUI Datepicker. For example:

```
form do |f|
  f.input :starts_at, as: :datepicker,
                      datepicker_options: {
                        min_date: "2013-10-8",
                        max_date: "+3D"
                      }

  f.input :ends_at, as: :datepicker,
                    datepicker_options: {
                      min_date: 3.days.ago.to_date,
```

```
                    max_date: "+1W +5D"
                  }
end
```

Displaying Errors

To display a list of all validation errors:

```
form do |f|
  f.semantic_errors *f.object.errors.keys
  # ...
end
```

This is particularly useful to display errors on virtual or hidden attributes.

Tabs

You can arrange content in tabs as shown below:

```
  form do |f|
    tabs do
      tab 'Basic' do
        f.inputs 'Basic Details' do
          f.input :email
          f.input :password
          f.input :password_confirmation
        end
      end

      tab 'Advanced' do
        f.inputs 'Advanced Details' do
          f.input :role
        end
      end
    end
    f.actions
  end
```

Customize the Create Another checkbox

In order to simplify creating multiple resources you may enable ActiveAdmin to show nice “Create Another” checkbox alongside of Create Model button. It may be enabled for the whole application:

```
ActiveAdmin.setup do |config|
  config.create_another = true
end
```

or for the particular resource:

```
ActiveAdmin.register Post do
  config.create_another = true
end
```

Customize the Show Page

The show block is rendered within the context of the view and uses Arbre syntax.

With the `show` block, you can render anything you want.

```
ActiveAdmin.register Post do
  show do
    h3 post.title
    div do
      simple_format post.body
    end
  end
end
```

You can render a partial at any point:

```
ActiveAdmin.register Post do
  show do
    # renders app/views/admin/posts/_some_partial.html.erb
    render 'some_partial', { post: post }
  end
end
```

If you'd like to keep the default AA look, you can use `attributes_table`:

```
ActiveAdmin.register Ad do
  show do
    attributes_table do
      row :title
      row :image do |ad|
        image_tag ad.image.url
      end
    end
    active_admin_comments
  end
end
```

You can also customize the title of the object in the show screen:

```
show title: :name do
  # ...
end
```

If you want a more data-dense page, you can combine a sidebar:

```
ActiveAdmin.register Book do
  show do
    panel "Table of Contents" do
      table_for book.chapters do
        column :number
        column :title
        column :page
      end
    end
    active_admin_comments
  end

  sidebar "Details", only: :show do
    attributes_table_for book do
      row :title
      row :author
      row :publisher
      row('Published?') { |b| status_tag b.published? }
    end
  end
end
```

Sidebar Sections

Sidebars allow you to put whatever content you want on the side the page.

```
sidebar :help do
  "Need help? Email us at help@example.com"
end
```

This will generate a sidebar on every page for that resource. The first argument is used as the title, and can be a symbol, string, or lambda.

You can also use Arbre to define HTML content.

```
sidebar :help do
  ul do
    li "Second List First Item"
    li "Second List Second Item"
  end
end
```

Sidebars can be rendered on a specific action by passing `:only` or `:except`.

```
sidebar :help, only: :index do
  "Need help? Email us at help@example.com"
end
```

If you want to conditionally display a sidebar section, use the :if option and pass it a proc which will be rendered within the view context.

```
sidebar :help, if: proc{ current_admin_user.super_admin? } do
  "Only for super admins!"
end
```

You can access your model as resource in the sidebar too:

```
sidebar :custom, only: :show do
  resource.a_method
end
```

You can also render a partial:

```
sidebar :help                     # app/views/admin/posts/_help_sidebar.html.erb
sidebar :help, partial: 'custom' # app/views/admin/posts/_custom.html.erb
```

It’s possible to add custom class name to the sidebar parent element by passing `class` option:

```
sidebar :help, class: 'custom_class'
```

By default sidebars are positioned in the same order as they defined, but it’s also possible to specify their position manually:

```
# will push Help section to the top (above default Filters section)
sidebar :help, priority: 0
```

Default sidebar priority is `10`.

Custom Controller Actions

Active Admin allows you to override and modify the underlying controller which is generated for you. There are helpers to add collection and member actions, or you can drop right in to the controller and modify its behavior.

Collection Actions

A collection action is a controller action which operates on the collection of resources. This method adds both the action to the controller as well as generating a route for you.

To add a collection action, use the collection_action method:

```
ActiveAdmin.register Post do

  collection_action :import_csv, method: :post do
    # Do some CSV importing work here...
    redirect_to collection_path, notice: "CSV imported successfully!"
  end

end
```

This collection action will generate a route at `/admin/posts/import_csv` pointing to the `Admin::PostsController#import_csv` controller action.

Member Actions

A member action is a controller action which operates on a single resource.

For example, to add a lock action to a user resource, you would do the following:

```
ActiveAdmin.register User do

  member_action :lock, method: :put do
    resource.lock!
    redirect_to resource_path, notice: "Locked!"
  end

end
```

This will generate a route at `/admin/users/:id/lock` pointing to the `Admin::UserController#lock` controller action.

HTTP Verbs

The `collection_action` and `member_action` methods both accept the `:method` argument to set the HTTP verb for the controller action and route.

Sometimes you want to create an action with the same name, that handles multiple HTTP verbs. In that case, this is the suggested approach:

```
member_action :foo, method: [:get, :post] do
  if request.post?
    resource.update_attributes! foo: params[:foo] || {}
    head :ok
  else
    render :foo
  end
end
```

Rendering

Custom controller actions support rendering within the standard Active Admin layout.

```
ActiveAdmin.register Post do

  # /admin/posts/:id/comments
  member_action :comments do
    @comments = resource.comments
    # This will render app/views/admin/posts/comments.html.erb
  end

end
```

If you would like to use the same view syntax as the rest of Active Admin, you can use the Arbre file extension: .arb.

For example, create `app/views/admin/posts/comments.html.arb` with:

```
table_for assigns[:post].comments do
  column :id
  column :author
  column :body do |comment|
    simple_format comment.body
  end
end
```

Page Titles

The page title for the custom action will be the translated version of the controller action name. For example, a member_action named “upload_csv” will look up a translation key of `active_admin.upload_csv`. If none are found, it defaults to the name of the controller action.

If this doesn’t work for you, you can always set the `@page_title` instance variable in your controller action to customize the page title.

```
ActiveAdmin.register Post do

  member_action :comments do
    @comments   = resource.comments
    @page_title = "#{resource.title}: Comments" # Sets the page title
  end

end
```

Action Items

To include your own action items (like the New, Edit and Delete buttons), add an `action_item` block. The first parameter is just a name to identify the action, and is required. For example, to add a “View on site” button to view a blog post:

```
action_item :view, only: :show do
  link_to 'View on site', post_path(post) if post.published?
end
```

Actions items also accept the `:if` option to conditionally display them:

```
action_item :super_action,
            only: :show,
            if: proc{ current_admin_user.super_admin? } do
  "Only display this to super admins on the show screen"
end
```

Modifying the Controller

The generated controller is available to you within the registration block by using the `controller` method.

```
ActiveAdmin.register Post do

  controller do
    # This code is evaluated within the controller class

    def define_a_method
      # Instance method
    end
  end

end
```

Batch Actions

By default, the index page provides you a “Batch Action” to quickly delete records, as well as an API for you to easily create your own. Note that if you override the default index, you must add `selectable_column` back for batch actions to be usable:

```
index do
  selectable_column
  # ...
end
```

Creating your own

Use the `batch_action` DSL method to create your own. It behaves just like a controller method, so you can send the client whatever data you like. Your block is passed an array of the record IDs that the user selected, so you can perform your desired batch action on all of them:

```
ActiveAdmin.register Post do
  batch_action :flag do |ids|
    batch_action_collection.find(ids).each do |post|
      post.flag! :hot
    end
    redirect_to collection_path, alert: "The posts have been flagged."
  end
end
```

Disabling Batch Actions

You can disable batch actions at the application, namespace, or resource level:

```
# config/initializers/active_admin.rb
ActiveAdmin.setup do |config|

  # Application level:
  config.batch_actions = false

  # Namespace level:
  config.namespace :admin do |admin|
    admin.batch_actions = false
  end
end

# app/admin/post.rb
ActiveAdmin.register Post do

  # Resource level:
  config.batch_actions = false
end
```

Modification

If you want, you can override the default batch action to do whatever you want:

```
ActiveAdmin.register Post do
  batch_action :destroy do |ids|
    redirect_to collection_path, alert: "Didn't really delete these!"
  end
end
```

Removal

You can remove batch actions by simply passing false as the second parameter:

```
ActiveAdmin.register Post do
  batch_action :destroy, false
```

```
end
```

Conditional display

You can control whether or not the batch action is available via the `:if` option, which is executed in the view context.

```
ActiveAdmin.register Post do
  batch_action :flag, if: proc{ can? :flag, Post } do |ids|
    # ...
  end
end
```

Priority in the drop-down menu

You can change the order of batch actions through the `:priority` option:

```
ActiveAdmin.register Post do
  batch_action :destroy, priority: 1 do |ids|
    # ...
  end
end
```

Confirmation prompt

You can pass a custom string to prompt the user with:

```
ActiveAdmin.register Post do
  batch_action :destroy, confirm: "Are you sure??" do |ids|
    # ...
  end
end
```

Batch Action forms

If you want to capture input from the user as they perform a batch action, Active Admin has just the thing for you:

```
batch_action :flag, form: {
  type: %w[Offensive Spam Other],
  reason: :text,
  notes:  :textarea,
  hide:   :checkbox,
  date:   :datepicker
} do |ids, inputs|
  # inputs is a hash of all the form fields you requested
  redirect_to collection_path, notice: [ids, inputs].to_s
end
```

If you pass a nested array, it will behave just like Formtastic would, with the first element being the text displayed and the second element being the value.

```
batch_action :doit, form: {user: [['Jake',2], ['Mary',3]]} do |ids, inputs|
  User.find(inputs[:user])
  # ...
end
```

When you have dynamic form inputs you can pass a proc instead:

```
batch_action :doit, form: -> { {user: User.pluck(:name, :id)} } do |ids, inputs|
  User.find(inputs[:user])
  # ...
end
```

Under the covers this is powered by the JS `ActiveAdmin.modal_dialog` which you can use yourself:

```
if $('body.admin_users').length
  $('a[data-prompt]').click ->
    ActiveAdmin.modal_dialog $(@).data('prompt'), comment: 'textarea',
      (inputs)=>
        $.post "/admin/users/#{$(@).data 'id'}/change_state",
          comment: inputs.comment, state: $(@).data('state'),
          success: ->
            window.location.reload()
```

Translation

By default, the name of the batch action will be used to lookup a label for the menu. It will lookup in `active_admin.batch_actions.labels.#{your_batch_action}`.

So this:

```
ActiveAdmin.register Post do
  batch_action :publish do |ids|
    # ...
  end
end
```

Can be translated with:

```
# config/locales/en.yml
en:
  active_admin:
    batch_actions:
      labels:
        publish: "Publish"
```

Support for other index types

You can easily use `batch_action` in the other index views, *Grid*, *Block*, and *Blog*; however, these will require custom styling to fit your needs.

```
ActiveAdmin.register Post do

  # By default, the "Delete" batch action is provided

  # Index as Grid
  index as: :grid do |post|
    resource_selection_cell post
    h2 auto_link post
  end

  # Index as Blog requires nothing special

  # Index as Block
  index as: :block do |post|
    div for: post do
      resource_selection_cell post
    end
  end

end
```

BTW

In order to perform the batch action, the entire *Table*, *Grid*, etc. is wrapped in a form that submits the IDs of the selected rows to your batch_action.

Since nested `<form>` tags in HTML often results in unexpected behavior, you may need to modify the custom behavior you've built using to prevent conflicts.

Specifically, if you are using HTTP methods like `PUT` or `PATCH` with a custom form on your index page this may result in your batch action being `PUT`ed instead of `POST`ed which will create a routing error. You can

get around this by either moving the nested form to another page or using a POST so it doesn’t override the batch action. As well, behavior may vary by browser.

Custom Pages

If you have data you want on a standalone page that isn't tied to a resource, custom pages provide you with a familiar syntax and feature set:

- a menu item
- sidebars
- action items
- page actions

Create a new Page

Creating a page is as simple as calling `register_page`:

```
# app/admin/calendar.rb
ActiveAdmin.register_page "Calendar" do
  content do
    para "Hello World"
  end
end
```

Anything rendered within `content` will be the main content on the page. Partials behave exactly the same way as they do for resources:

```
# app/admin/calendar.rb
ActiveAdmin.register_page "Calendar" do
  content do
    render partial: 'calendar'
  end
end

# app/views/admin/calendar/_calendar.html.arb
table do
  thead do
    tr do
      %w[Sunday Monday Tuesday Wednesday Thursday Friday Saturday].each &method(:th)
    end
  end
  tbody do
    # ...
  end
end
```

Customize the Menu

See the Menu documentation.

Customize the breadcrumbs

```
ActiveAdmin.register_page "Calendar" do
  breadcrumb do
    ['admin', 'calendar']
  end
end
```

Customize the Namespace

We use the `admin` namespace by default, but you can use anything:

```
# Available at /today/calendar
ActiveAdmin.register_page "Calendar", namespace: :today
```

```
# Available at /calendar
ActiveAdmin.register_page "Calendar", namespace: false
```

Belongs To

To nest the page within another resource, you can use the `belongs_to` method:

```
ActiveAdmin.register Project
ActiveAdmin.register_page "Status" do
  belongs_to :project
end
```

See also the Belongs To documentation and examples.

Add a Sidebar

See the Sidebars documentation.

Add an Action Item

Just like other resources, you can add action items. The difference here being that `:only` and `:except` don't apply because there's only one page it could apply to.

```
action_item :view_site do
  link_to "View Site", "/"
end
```

Add a Page Action

Page actions are custom controller actions (which mirror the resource DSL for the same feature).

```
page_action :add_event, method: :post do
  # ...
  redirect_to admin_calendar_path, notice: "Your event was added"
end

action_item :add do
  link_to "Add Event", admin_calendar_add_event_path, method: :post
end
```

This defines the route `/admin/calendar/add_event` which can handle HTTP POST requests.

Clicking on the action item will reload page and display the message "Your event was added"

Page actions can handle multiple HTTP verbs.

```
page_action :add_event, method: [:get, :post] do
  # ...
end
```

See also the Custom Actions example.

Decorators

Active Admin allows you to use the decorator pattern to provide view-specific versions of a resource. Draper is recommended but not required.

To use decorator support without Draper, your decorator must support a variety of collection methods to support pagination, filtering, etc. See this github issue discussion and this gem for more details.

Example usage

```
# app/models/post.rb
class Post < ActiveRecord::Base
  # has title, content, and image_url
end

# app/decorators/post_decorator.rb
class PostDecorator < Draper::Decorator
  delegate_all

  def image
    h.image_tag model.image_url
  end
end

# app/admin/post.rb
ActiveAdmin.register Post do
  decorate_with PostDecorator

  index do
    column :title
    column :image
    actions
  end
end
```

Forms

By default, ActiveAdmin does *not* decorate the resource used to render forms. If you need ActiveAdmin to decorate the forms, you can pass `decorate: true` to the form block.

```
ActiveAdmin.register Post do
  decorate_with PostDecorator

  form decorate: true do |f|
    # ...
  end
end
```

Arbre Components

Arbre allows the creation of shareable and extendable HTML components and is used throughout Active Admin to create view components.

Text Node

Sometimes it makes sense to insert something into a registered resource like a non-breaking space or some text. The text_node method can be used to insert these elements into the page inside of other Arbre components or resource controller functions.

```
ActiveAdmin.register Post do
  show do
    panel "Post Details" do
      attributes_table_for post do
        row :id
        row 'Tags' do
          post.tags.each do |tag|
            a tag, href: admin_post_path(q: {tagged_with_contains: tag})
            text_node " ".html_safe
          end
        end
      end
    end
  end
end
```

Panels

A panel is a component that takes up all available horizontal space and takes a title and a hash of attributes as arguments. If a sidebar is present, a panel will take up the remaining space.

This will create two stacked panels:

```
show do
  panel "Post Details" do
    render partial: "details", locals: {post: post}
  end

  panel "Post Tags" do
    render partial: "tags",    locals: {post: post}
  end
end
```

Columns

The Columns component allows you draw content into scalable columns. All you need to do is define the number of columns and the component will take care of the rest.

Simple Columns

To create simple columns, use the `columns` method. Within the block, call the #column method to create a new column.

```
columns do
  column do
    span "Column #1"
  end

  column do
    span "Column #2"
  end
```

```
end
```

Spanning Multiple Columns

To create columns that have multiple spans, pass the :span option to the column method.

```
columns do
  column span: 2 do
    span "Column # 1"
  end
  column do
    span "Column # 2"
  end
end
```

By default, each column spans 1 column. The above layout would have 2 columns, the first being twice as large as the second.

Custom Column Widths

Active Admin uses a fluid width layout, causing column width to be defined using percentages. Due to using this style of layout, columns can shrink or expand past points that may not be desirable. To overcome this issue, columns provide `:max_width` and `:min_width` options.

```
columns do
  column max_width: "200px", min_width: "100px" do
    span "Column # 1"
  end
  column do
    span "Column # 2"
  end
end
```

In the above example, the first column will not grow larger than 200px and will not shrink less than 100px.

Table For

Table For provides the ability to create tables like those present in `index_as_table`. It takes a collection and a hash of options and then uses `column` to build the fields to show with the table.

```
table_for order.payments do
  column(:payment_type) { |payment| payment.payment_type.titleize }
  column "Received On",     :created_at
  column "Details & Notes", :payment_details
  column "Amount",          :amount_in_dollars
end
```

the `column` method can take a title as its first argument and data (`:your_method`) as its second (or first if no title provided). Column also takes a block.

Status tag

Status tags provide convenient syntactic sugar for styling items that have status. A common example of where the status tag could be useful is for orders that are complete or in progress. `status_tag` takes a status, like “In Progress”, and a hash of options. The status_tag will generate HTML markup that Active Admin CSS uses in styling.

```
status_tag 'In Progress'
# => <span class='status_tag in_progress'>In Progress</span>

status_tag 'active', class: 'important', id: 'status_123', label: 'on'
# => <span class='status_tag active important' id='status_123'>on</span>
```

Tabs

The Tabs component is helpful for saving page real estate. The first tab will be the one open when the page initially loads and the rest hidden. You can click each tab to toggle back and forth between them. Arbre supports unlimited number of tabs.

```
tabs do
  tab :active do
    table_for orders.active do
      ...
    end
  end

  tab :inactive do
    table_for orders.inactive do
      ...
    end
  end
end
```

Authorization Adapter

Active Admin offers the ability to define and use your own authorization adapter. If implemented, the '#authorized?' will be called when an action is taken. By default, '#authorized?' returns true.

Setting up your own AuthorizationAdapter

Setting up your own `AuthorizationAdapter` is easy! The following example shows how to set up and tie your authorization adapter class to Active Admin:

```
# app/models/only_authors_authorization.rb
class OnlyAuthorsAuthorization < ActiveAdmin::AuthorizationAdapter

  def authorized?(action, subject = nil)
    case subject
    when normalized(Post)
      # Only let the author update and delete posts
      if action == :update || action == :destroy
        subject.author == user
      else
        true
      end
    else
      true
    end
  end

end
```

In order to hook up `OnlyAuthorsAuthorization` to Active Admin, go to your application's `config/initializers/active_admin.rb` and add/modify the line:

```
config.authorization_adapter = "OnlyAuthorsAuthorization"
```

Authorization adapters can be configured per ActiveAdmin namespace as well, for example:

```
ActiveAdmin.setup do |config|
  config.namespace :admin do |ns|
    ns.authorization_adapter = "AdminAuthorization"
  end
  config.namespace :my do |ns|
    ns.authorization_adapter = "DashboardAuthorization"
  end
end
```

Now, whenever a controller action is performed, the `OnlyAuthorsAuthorization`'s `#authorized?` method will be called.

Getting Access to the Current User

From within your authorization adapter, you can call the `#user` method to retrieve the current user.

```
class OnlyAdmins < ActiveAdmin::AuthorizationAdapter

  def authorized?(action, subject = nil)
    user.admin?
  end

end
```

Scoping Collections in Authorization Adapters

`ActiveAdmin::AuthorizationAdapter` also provides a hook method (`#scope_collection`) for the

adapter to scope the resource's collection. For example, you may want to centralize the scoping:

```
class OnlyMyAccount < ActiveAdmin::AuthorizationAdapter

  def authorized?(action, subject = nil)
    subject.account == user.account
  end

  def scope_collection(collection, action = Auth::READ)
    collection.where(account_id: user.account_id)
  end

end
```

All collections presented on Index Screens will be passed through this method and will be scoped accordingly.

Managing Access to Pages

Pages, just like resources, get authorized too. When authorizing a page, the subject will be an instance of `ActiveAdmin::Page`.

```
class OnlyDashboard < ActiveAdmin::AuthorizationAdapter
  def authorized?(action, subject = nil)
    case subject
    when ActiveAdmin::Page
      action == :read &&
        subject.name == "Dashboard" &&
        subject.namespace.name == :admin
    else
      false
    end
  end
end
```

Action Types

By default Active Admin simplifies the controller actions into 4 actions:

- `:read` - This controls if the user can view the menu item as well as the index and show screens.
- `:create` - This controls if the user can view the new screen and submit the form to the create action.
- `:update` - This controls if the user can view the edit screen and submit the form to the update action.
- `:destroy` - This controls if the user can delete a resource.

Each of these actions is available as a constant. Eg: `:read` is available as `ActiveAdmin::Authorization::READ`.

Checking for Authorization in Controllers and Views

Active Admin provides a helper method to check if the current user is authorized to perform an action on a subject.

Simply use the `#authorized?(action, subject)` method to check.

```
ActiveAdmin.register Post do

  index do
    column :title
    column '' do |post|
      link_to 'Edit', admin_post_path(post) if authorized? :update, post
    end
  end
```

```
end
```

If you are implementing a custom controller action, you can use the `#authorize!` method to raise an `ActiveAdmin::AccessDenied` exception.

```
ActiveAdmin.register Post do

  member_action :publish, method: :post do
    post = Post.find(params[:id])

    authorize! :publish, post
    post.publish!

    flash[:notice] = "Post has been published"
    redirect_to [:admin, post]
  end

  action_item :publish, only: :show do
    if !post.published? && authorized?(:publish, post)
      link_to "Publish", publish_admin_post_path(post), method: :post
    end
  end

end
```

Using the CanCan Adapter

Sub-classing `ActiveAdmin::AuthorizationAdapter` is fairly low level. Many times it's nicer to have a simpler DSL for managing authorization. Active Admin provides an adapter out of the box for CanCan and CanCanCan.

To use the CanCan adapter, simply update the configuration in the Active Admin initializer:

```
config.authorization_adapter = ActiveAdmin::CanCanAdapter
```

You can also specify a method to be called on unauthorized access. This is necessary in order to prevent a redirect loop that can happen if a user tries to access a page they don't have permissions for (see #2081).

```
config.on_unauthorized_access = :access_denied
```

The method `access_denied` would be defined in `application_controller.rb`. Here is one example that redirects the user from the page they don't have permission to access to a resource they have permission to access (organizations in this case), and also displays the error message in the browser:

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  def access_denied(exception)
    redirect_to admin_organizations_path, alert: exception.message
  end
end
```

By default this will use the ability class named "Ability". This can also be changed from the initializer:

```
config.cancan_ability_class = "MyCustomAbility"
```

Now you can simply use CanCan or CanCanCan the way that you would expect and Active Admin will use it for authorization:

```
# app/models/ability.rb
class Ability
  include CanCan::Ability

  def initialize(user)
    can :manage, Post
    can :read, User
    can :manage, User, id: user.id
```

```
    can :read, ActiveAdmin::Page, name: "Dashboard", namespace_name: :admin
  end

end
```

To view more details about the API's, visit project pages of CanCan and CanCanCan.

Using the Pundit Adapter

Active Admin provides an adapter out of the box also for Pundit.

To use the Pundit adapter, simply update the configuration in the Active Admin initializer:

```
config.authorization_adapter = ActiveAdmin::PunditAdapter
```

You can simply use Pundit the way that you would expect and Active Admin will use it for authorization. Check Pundit's documentation to set up Pundit in your application. If you want to use batch actions just ensure that `destroy_all?` method is defined in your policy class. You can use this template policy in your application instead of default one generated by Pundit's `rails g pundit:install` command.