rails g model company name description:text contactname street city postcode country phone email homepage user:references

rails g model logo filename content\_type file\_contents:binary description:text user:references company:references

rails g model car make model year:datetime body version classification type door:integer seat:integer engineplace drivetrain company:references

rails g scaffold\_controller car make model year:datetime body version classification type door:integer seat:integer engineplace drivetrain

rails g model photo filename content\_type file\_contents:binary description:text car:references bike:references carpart:references bikepart:references

rails g scaffold\_controller photo file description

rails g scaffold\_controller company name description:text contactname street city postcode country phone email homepage

rails g migration rename\_type\_to\_bodytype

rails g model carpart name code description detail price:float stock:integer reserved:integer reservedtime:datetime paidtime:datetime sold:integer company:references car:references user:references

rails g scaffold\_controller carpart name code description detail price:float stock:integer reserved:integer reservedtime:datetime paidtime:datetime sold:integer

rails g migration add\_column\_reference\_to\_company company:references

rails g migration change\_detail\_name\_and\_type\_in\_carparts

\_\_\_\_\_\_\_\_\_\_\_\_

**1.** Make sure you include confirmable in Model.devise call

class User < ActiveRecord::Base

devise :database\_authenticatable, :confirmable ...

end

**2.** Make sure you add confirmable to the user migration

create\_table :users do |t|

t.database\_authenticatable

t.confirmable

...

end

If you're using devise 2.0+ this fails because devise no longer provides migration helpers, and so t.confirmable raises an error. Instead, copy the block labeled "Confirmable" from [their migration guide](https://github.com/plataformatec/devise/wiki/How-To:-Upgrade-to-Devise-2.0-migration-schema-style#after).

## Confirmable

# t.string :confirmation\_token

# t.datetime :confirmed\_at

# t.datetime :confirmation\_sent\_at

# t.string :unconfirmed\_email # Only if using reconfirmable

rails g migration add\_confirmable\_to\_users

**3.** Generate the devise views, with either of the following commands,so you can override the devise mailer views:

rails generate devise:views # global

rails generate devise:views users # scoped

You can now override the mailer views in devise/mailer/confirmation\_instructions.html.erb or users/mailer/confirmation\_instructions.html.erb depending on your setup

**4.** For **development** environment add the following config lines in /config/environments/development.rb

config.action\_mailer.default\_url\_options = { :host => 'localhost:3000' }

config.action\_mailer.delivery\_method = :smtp

config.action\_mailer.smtp\_settings = {:address => "localhost", :port => 1025}

**5.** For **production** environment in /config/environments/production.rb you may use something similar to the following (supposing you have a SMTP server on localhost:25):

config.action\_mailer.default\_url\_options = {:host => 'yourdomain.com'}

config.action\_mailer.delivery\_method = :smtp

config.action\_mailer.smtp\_settings = {

:address => "127.0.0.1",

:port => 25,

:domain => 'yourdomain.com'

}

**6** To test the setup in development install the mailcatcher gem, that you will use as a SMTP server in development, catching all incoming mails and displaying them on http://localhost:1080/:

gem install mailcatcher

Once installed start the mailcatcher server with the command:

mailcatcher

A toy SMTP server will be running on port 1025 catching emails and displaing them on HTTP port 1080.

**You can now create an account and see the confirmations.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

You basically want to use AJAX to display a customer's details. For this you can use remote: trueoption provided by rails for the link\_to helper. What we are going to do next :

1. Create a div that will contain the loaded data. In this case div#current\_customer
2. Add remote: true to your link :
3. <td><%= link\_to 'Show', customer, remote: true %></td>
4. Name your partial customers/\_show.html.erb (don't forget the \_so it can be called as a partial) :
5. <p>
6. <strong>Name:</strong>
7. <%= @customer.name %>
8. <%= @customer.id %>
9. </p>
10. <%= link\_to 'Edit Customer', edit\_customer\_path(@customer) %> |
11. <%= link\_to 'Back', customers\_path %> # You should remove this link
12. Respond to Javascript in the show method in CustomersController :
13. respond\_to do |format|
14. format.js {render layout: false} # Add this line to you respond\_to block
15. end
16. Create your show.js.erb view, which is going to handle the front-end changes when respond\_to :jsis gonna be called (In this case triggered by remote: true)
17. Tell show.js.erb what it must do (Replace #current\_customer content with your partial, using the right @customer) :

customers/show.js.erb

$("#current\_customer").html("<%= escape\_javascript(render partial: 'customers/show', locals: { customer: @customer } ) %>");

customers/index.html.erb

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Add tinymce-rails to your Gemfile**

gem 'tinymce-rails'

**2. Create a config/tinymce.yml file with your global configuration options:**

toolbar:

- styleselect | bold italic | undo redo

- image | link

plugins:

- image

- link

To define multiple configuration sets, follow this syntax (a default configuration must be specified):

default:

plugins:

- image

- link

alternate:

selector: textarea.table-editor

toolbar: styleselect | bold italic | undo redo | table

plugins:

- table

**3. Include the TinyMCE assets**

Use *one* of the following options to include TinyMCE assets.

(1) Add to your application.js:

//= require tinymce

or (2) with jQuery integration:

//= require tinymce-jquery

**4. Initialize TinyMCE**

For each textarea that you want to use with TinyMCE, add the "tinymce" class and ensure it has a unique ID:

<%= text\_area\_tag :content, "", :class => "tinymce", :rows => 40, :cols => 120 %>

or if you are using Rails' form builders:

<%= f.text\_area :content, :class => "tinymce", :rows => 40, :cols => 120 %>

Then invoke the tinymce helper to initialize TinyMCE:

<%= tinymce %>

Custom options can be passed to tinymce to override the global options specified in config/tinymce.yml:

<%= tinymce :theme => "simple", :language => "de", :plugins => ["wordcount", "paste"] %>

Alternate configurations defined in 'config/tinymce.yml' can be used with:

<%= tinymce :alternate %>

## Asset Compilation

Since TinyMCE loads most of its files dynamically, some workarounds are required to ensure that the TinyMCE asset files are accessible using non-digested filenames.

As of tinymce-rails 3.5.11, 4.1.10 and 4.2.1, two alternative asset installation methods are available, which can be changed by setting config.tinymce.install within your config/application.rb file. These methods are called when you run rake asset:precompile (via Rake::Task#enhance) after the regular application assets are compiled.

The default method (as of 4.5.2), compile, adds the TinyMCE paths to the Sprockets precompilation paths and then creates symlinks from the non-digested filenames to their digested versions.

config.tinymce.install = :compile

If you experience issues with the compile method, you may wish to use the copy method instead, which copies the TinyMCE assets directly into public/assets and appends the file information into the asset manifest. The copy\_no\_preserve method is also available of you do not wish to or cannot preserve file modes on your filesystem.

config.tinymce.install = :copy

If you are including TinyMCE via application.js or using the tinymce\_assets helper, you do not need to manually alter the precompile paths. However if you wish to include tinymce-jquery.js independently (i.e. using javascript\_include\_tag), you will need to add it to the precompile list in config/environments/production.rb:

config.assets.precompile << "tinymce-jquery.js"

\_\_\_\_\_\_\_\_\_\_\_

## Saving arrays, hashes, and other non-mappable objects in text columns

Active Record can serialize any object in text columns using YAML. To do so, you must specify this with a call to the class method [serialize](http://api.rubyonrails.org/classes/ActiveRecord/AttributeMethods/Serialization/ClassMethods.html#method-i-serialize). This makes it possible to store arrays, hashes, and other non-mappable objects without doing any additional work.

**class User < ActiveRecord::Base**

serialize :preferences

**end**

user = User.create(preferences: { "background" => "black", "display" => large })

User.find(user.id).preferences *# => { "background" => "black", "display" => large }*

You can also specify a class option as the second parameter that'll raise an exception if a serialized object is retrieved as a descendant of a class not in the hierarchy.

**class User < ActiveRecord::Base**

serialize :preferences, Hash

**end**

user = User.create(preferences: %w( one two three ))

User.find(user.id).preferences *# raises SerializationTypeMismatch*

When you specify a class option, the default value for that attribute will be a new instance of that class.

**class User < ActiveRecord::Base**

serialize :preferences, OpenStruct

**end**

user = User.new

user.preferences.theme\_color = "red"

First, definitely check out ActiveRecord serialize and see if it does what you need: <http://api.rubyonrails.org/classes/ActiveRecord/Base.html#method-c-serialize>

However, if you need JSON specifically (serialize uses YAML by default), then you can always fake it by hand.

You can simply build a hash in Ruby and then call to\_json on it before assigning it to your model attribute.

data = { 'photoid' => 123, 'photoname' => "asdasd", 'creator\_id' => "asdasd" }

myrecord.stored\_data = data.to\_json

myrecord.save

<%= content\_tag :tr do%>

<% @carpart.details.to\_h.each do |key, value| %>

<%= content\_tag :th, "#{key}:" %>

<%= content\_tag :td, "#{value}" %>

<% end %>