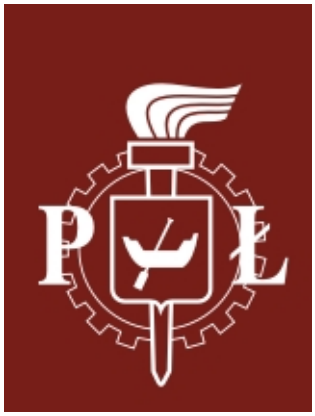
Technical University of Lodz

International Faculty of Engineering and
Faculty of Electrical, Electronic, Computer
and Control Engineering



# Plug-in application for automatic generation of administration panel

## Aplikacja do automatycznego tworzenia panelu administracyjnego

*Author:*
Piotr Jakubowski

*Under the supervision of:*
Dr inż. Andrzej Romanowski

*In cooperation with:*
Ragnarson Sp. z o.o.

Lodz, 2011

**Abstract**

The aim of the thesis is to design and implement a plugin application for Ruby on Rails framework. The plugin would provide the functionality of automatic generation of administration panel for managing resources (models) of the web application. The idea behind the project is that it should be easy to use, without the need of interfering with application code.

The deliverables of the thesis are both theoretical description and working code of the plugin application. The theoretical work covers the background of technologies used for the purpose of the project as well as it describes the process of how the plugin has been implemented.

**Streszczenie**

Celem pracy magisterskiej jest stworzenie oprogramowania pozwalającego na automatyczne tworzenie panelu administracyjnego do zarządzania modelami aplikacji webowej w środowisku Ruby on Rails. Głównym założeniem projektu jest łatwość użycia oraz możliwości uruchomienia bez zbędnego zmieniania kodu istniejącej aplikacji.

Efektem pracy jest zarówno teoretyczny opis jak i kod działającego pluginu. Praca teoretyczna opisuje zarówno technologie wykorzystane przy tworzeniu projektu jak i proces implementacji.

*Acknowledgments*

# Contents

# List of Figures

# Chapter 1

# Introduction

Along with the huge popularization of the Internet there appeared enormous market for all sorts of web applications. Every month big companies, as well as small startups, try to get as much of this market by bringing in new features and shocking the crowd with innovative functionalities. Surprisingly, the market seems to be far from saturation as brilliant ideas except for fulfilling known demand create new needs as well.

New tools started to emerge in order to meet the needs of Internet population. Dynamic languages started to supersede old and known. Python and Ruby became the new Java and C. The speed of development surpassed the speed of execution as technical limitations and price of servers are not the issue any more. Moreover, web development frameworks made it possible to code the application on the new level of abstraction - we can instantaneously start to implement our business logic, without the need to think about how we should persist our entities into database and how we should handle requests and respond to them. Java developers got Spring and Hibernate, but the noteworthy Ruby on Rails took web development to the whole new level. The threshold of making the ideas happen has been successfully shrunken.

As mentioned before, today's software development has been focused on the effectiveness and speed of development. A lot of work has been done to create developer-oriented tools that try to make developing business ideas effortless.

Of course, there is plenty of room for improvements. One of them is the idea of automating process of creating the administration panel for the application. Very often administration part of the application is mundane collection of CRUD[1] actions. It is the "must do" part.

[1] Create Read Update Delete

User-oriented features is the target thing of development. This is what makes money and keeps us all employed. It seems like possibility to be able to automatically generate panel for administering our resources would make development of web application even more dynamic. It would bring plenty of benefits, such as:

- Whole team can focus on what really matters, which is creating functionality that would be an added-value for users of the application
- Adding new features and resources to the application would require less effort - as soon as the user-oriented functionality is ready the administration part is automatically generated.
- Business people connected with the project can instantly browse new resources in the application.

Thereby, the aim of this thesis is to create and demonstrate the process of creation of plugin that would enable automatic generation of administration part of the application. The plugin would be developed for Ruby on Rails framework mentioned before.

# Chapter 2

# Market Research

Thanks to its rapidly growing popularity, Ruby and Ruby on Rails got very broad community. Not only does it result in a huge number of places you can ask for help, but also in all kind of libraries that would make the life of web developer easier. And as it seems I am not alone stating the issue described above. There are already several implementations attacking the problem of admin interface. Some of them are more successful than others, there are different kinds of approaches and different outcomes.

In order to see what market has to offer I looked through available solutions and chose 4 most popular ones in order to see what kind of functionalities they provide and what flaws they suffer from.

## 2.1 Active Scaffold[1]

According to Ruby Toolbox[2] it is the most popular plugin for admin interface generation. Unfortunately, it seems like the days of glory for this gem are long gone. The plugin does not support Rails in version 3, which despite being released very recently already became a standard for new applications.

Technicalities aside, it seems to be quite functional. The almost out-of-the-box look is presented on figures 2.1 and 2.2

Active Scaffold ships with comprehensive API for customizing its behavior. User can change which columns are visible in which view or how the system behaves during particular actions. Moreover, it uses Asynchronous JavaScript and XML (AJAX) for interaction with user which gives nice and responsive user interface.

[1] http://activescaffold.com/
[2] http://ruby-toolbox.com/categories/rails_admin_interfaces.html

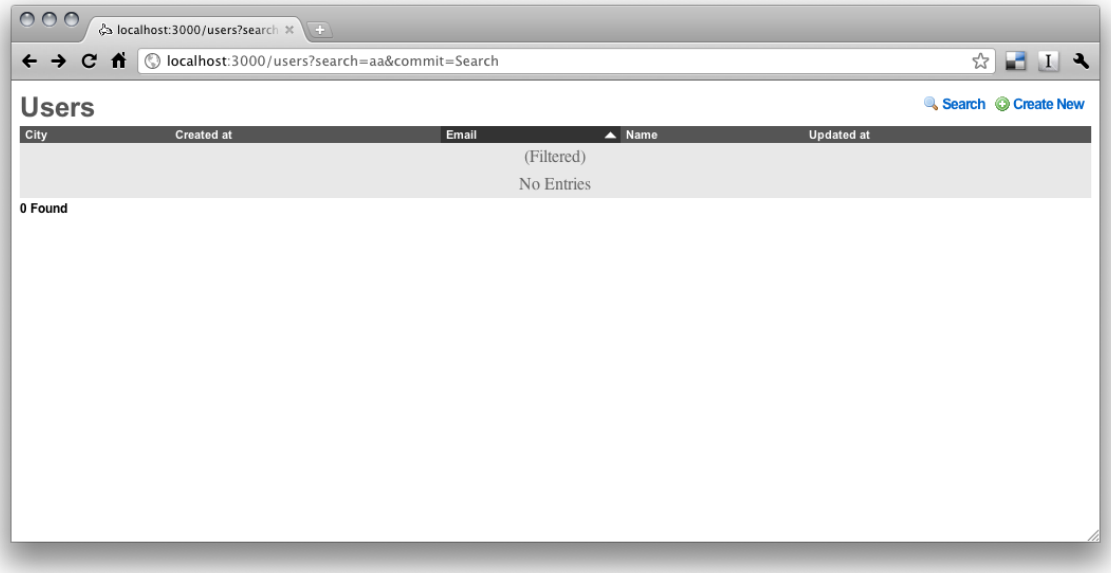



Figure 2.1: Active Scaffold

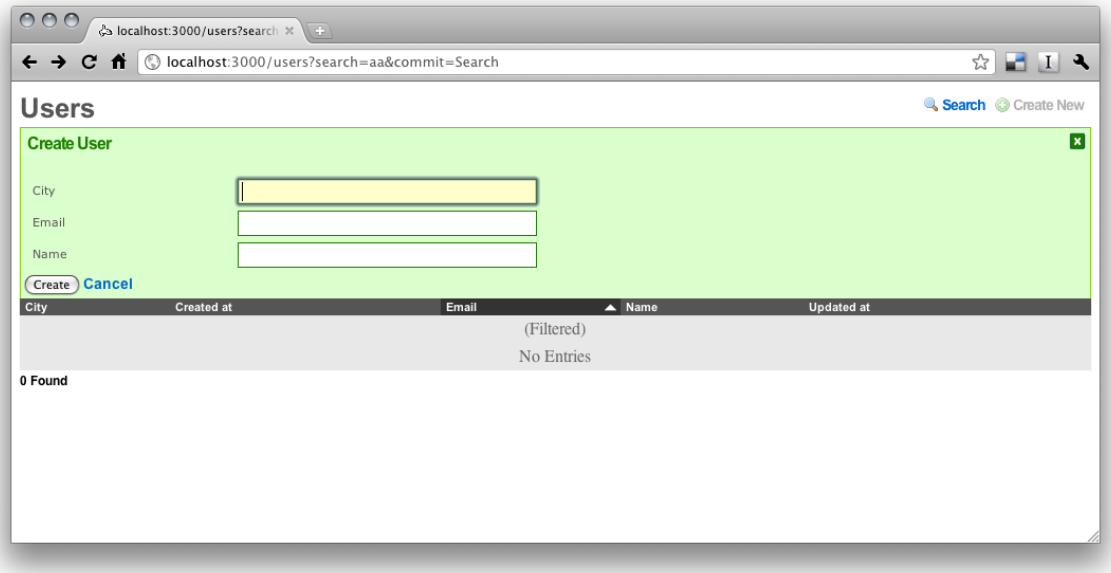



Figure 2.2: Automatically generated form in Active Scaffold

Unfortunately, it has big disadvantage. It is not entirely automatic. It requires developer to take following action to set it up:

- Add command to the layout of the application that will include styles and javascript for Active Scaffold.
- For every resource he wants to administer, he needs to create controller and add code that will set it up.

- Set up url for Active Scaffold actions for given controller by adding code to routes file.

This results in following inconveniences:

- Active Scaffold mixes with our application code.
- Every time we create some model we need to take care to set up Active Scaffold for it.

It seems like Active Scaffold instead of being "stand-alone" solution for administration is a set of commands, that speeds up set up of administration interface. It gives flexibility, but is not as hassle-free as could be and as developers would like it to be.

## 2.2 Typus[3]

Typus is reported to be the second most popular Rails admin interface generator. As opposed to Active Scaffold, its development team is still active. Thanks to that, it has been adapted to Rails 3 already.

Installation and set up is quick and effortless. We just need to install Typus and run one command. That way we get to the /admin path in our application. Its user interface is presented on figures 2.3 and 2.4.

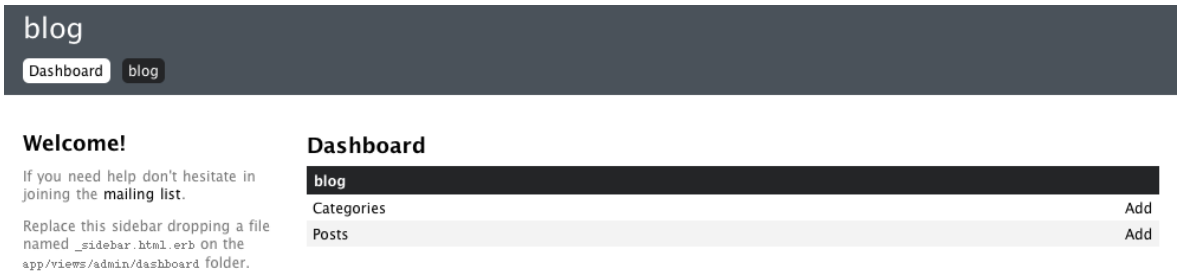



Figure 2.3: Typus main page

Documentation provides thorough description of possibilities of Typus' customization. We can decide which columns will be visible, how admins can search our models and so on. Most of the configuration is done by specifying appropriate YAML files. In addition, Typus is supposed to handle file uploads in your application, which is very convenient. However, it will happen only if your application is using particular plugin for files upload called Paperclip.

[3]http://core.typuscms.com/


Figure 2.4: Automatically generated form in Typus

But again, similiar as it was in Active Scaffold, Typus does not work entirely "on-the-fly". The generator we have to run in order to set Typus up, except for copying needed HTML, javascript and image files, creates controllers that serve particular Models of our application. As said before, this means that we can get full control on how given controller behaves - we can override Typus implementation with our own - but such approach is not as dynamic in terms of adapting to upcoming resources in the application. Of course, now it is just the sake of running simple command that would take care of everything once we introduce some new Model into application, but surely it is not something we just install and can forget about.

## 2.3 admin_data

Admin_data is next one on the list. Again, the project seems to be still maintained and has been upgraded to Rails 3 already. Installation is very much alike to the one of Typus. But, after installation we do not need to run any generators or create any kind of files. We instantly get access to /admin_data path and figure 2.5 presents what more or less we would see after trying it out.

Of course, it provides us with some level of customizability, though it is not as wide as in two other projects. But, admin_data has the level of dynamism I have been looking for. It creates the list of models on the fly, so while developing our application we

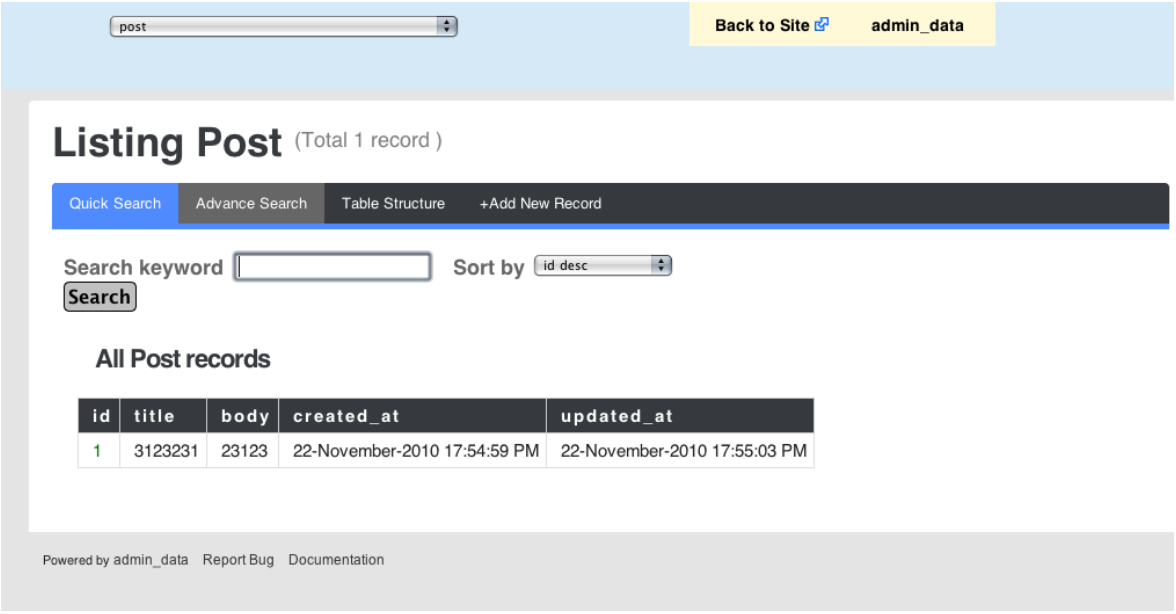



Figure 2.5: Admin data

will constantly have the most current version of our models lists, no question asked. Moreover, we can also see table's structure for given model, which may come in handy at times.

On the other hand, you can clearly see that admin_data has been created with programmers in mind. The user interface can be very hard to comprehend for people that do not know what is SQL or do not care how developers organized tables in the database. Therefore, it could be very hard to convince our business people that it is something they can use.

## 2.4 rails_admin

Although rails_admin is not present on the ruby-toolbox list it has been very popular lately. It has been developed during Ruby Summer of Code[4]. It has been developed strictly for Rails 3 so it uses all the best features and the best practices from the newest version of the framework.

Installation is very quick, although we need to go through few steps. Moreover, it depends on devise[5] gem, which takes care of users authentication, as rails_admin's default option is to make people to log in to the administration panel.

[4]http://rubysoc.org/
[5]https://github.com/plataformatec/devise

What we get after installation is very nice and user-friendly panel(figure 2.6). In addition, it tracks changes in records, what gives us functionality of seeing whole history of activities on records as well as people that made them. Additionally, we have that data presented in form of nice graph that gives us an idea of what was happening in the system throughout the month or year.

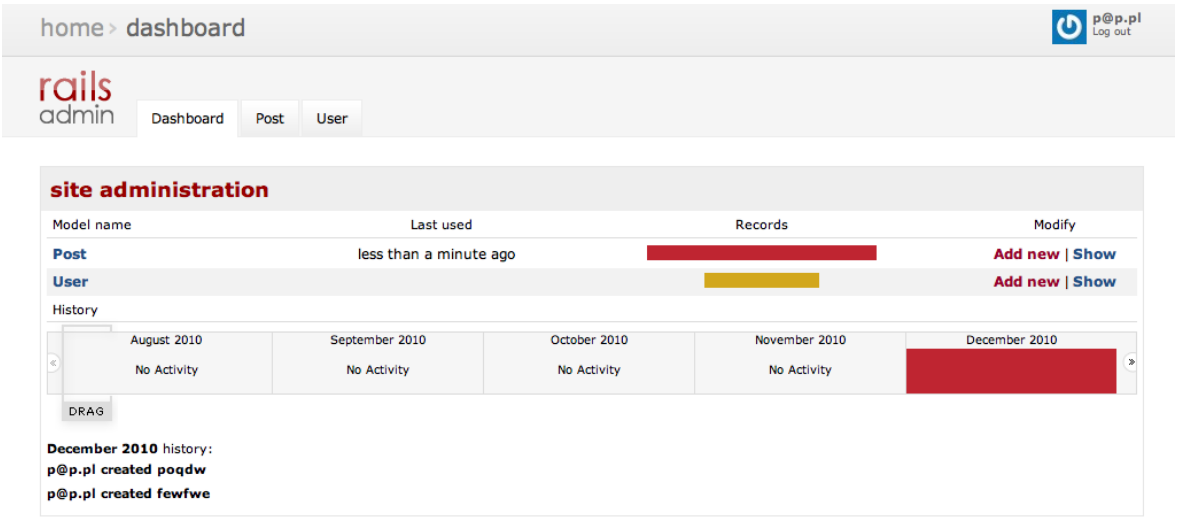



Figure 2.6: rails_admin dashboard

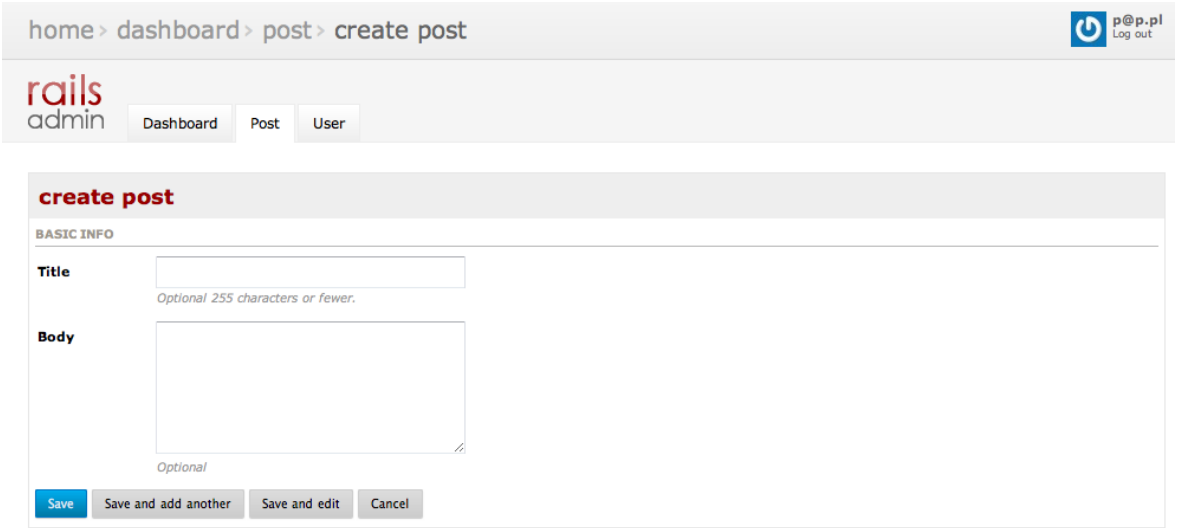



Figure 2.7: Form generated in rails_admin

Rails_admin seems to have real potential and although I have not seen it in use in production yet, soon it maybe become real power of Rails applications. Definitely, ideas used in that project are very interesting and may be very useful.

# Chapter 3

# Project definition

This chapter aims to state the scope of the project. Moreover, I would like to draw an outline of expected output and point out challenges that I would have to face during the development.

## 3.1 Scope of the project

As stated in the introduction, generation of administration panel seems to be an actual problem. Therefore, my aim is to develop a plugin or rather a gem[1] for Ruby language that would provide functionality of administration panel generation for Ruby on Rails application.

One of the first assumptions was that the installation of the administration panel should be effortless and should not require any changes to the code of application. Moreover, the administration panel should detect changes to business models "on-the-fly" and should adjust to reflect those changes in the panel without need of running any generators or other scripts.

The functionality of the administration panel would consist of:

- Listing all of business models in the application.
- For particular model listing all of the records.
- Creation and edition of records using automatically generated forms.
- Deletion of records.
- Proper handling of associations between models.

[1] TODO: More on gems in chapter bla bla bla

- Configuration of the panel regarding which models or model fields should be visible.

## 3.2 Project outline

The project would be developed for test application that would be a simple blog system. The functionality of the test application would be a subject of adjustments in order to check and prove appropriate functioning of the plugin.

To start with, test application would allow following actions:

- Add, edit, list and delete posts
- Add, edit, list and delete categories
- Assign posts to appropriate categories

Target functionality of plugin has been stated in previous section. Mockups of user interface of the plugin are depicted in figures 3.1, 3.2 and 3.3


Figure 3.1: Mockup of dashboard page of the plugin

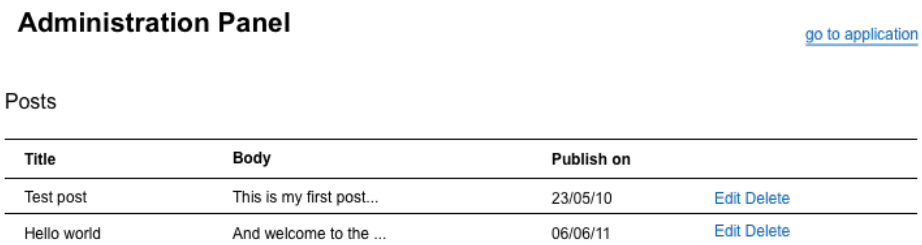



Figure 3.2: Mockup of page for listing records of given model

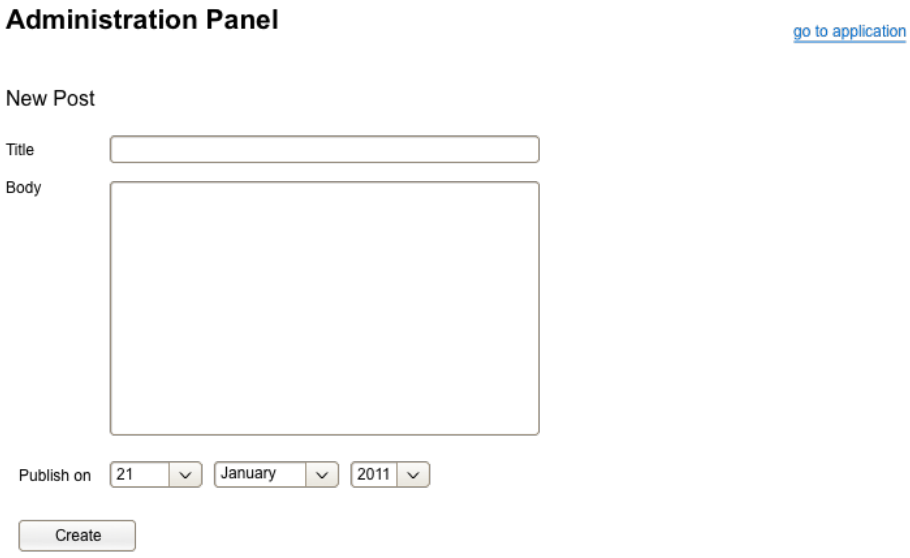



Figure 3.3: Example page for creation or edition of model record

## 3.3 Anticipated challenges

The challenge that is most striking while approaching this project is the fact that the solution has to be universal and totally independent of any domain specific workarounds. As a plugin, it has to be usable for all or at least majority of web applications. Therefore, it is necessary to take into consideration all kinds of association between models and all types of fields that may appear in those models. Moreover, I would have to watch out not to violate any business logic that some developer put in his models or not to introduce data inconsistency. Fortunately, well-design systems would rather fail raisin an exception than allow to perform some illegal action on models.

Furthermore, it seems like a lot of code in this project would need to figure out actions itself during the runtime basing on what kind of code would be in the application. In order to make that plugin functional, it would be necessary to dive into and make use of meta-programming, which may not be as easy to use, clear and readable. Fortunately, Ruby from its nature of being a dynamic language allows many meta-programming actions and provides developers with very pleasant reflection API.

Hopefully, those challenges would get overcome during design and development of the project in order to produce plugin that would make life of web programmers slightly easier.

# Chapter 4

# Technologies

The aim of this chapter is to present the environment in which the plugin will be developed. I will present, firstly, the features of Ruby language and then what is Ruby on Rails, how it is constructed and how we can enrich it with plugins. In the end I will take a look into possibilities of metaprogramming in Ruby and Rails.

## 4.1 Ruby

Here you will find information abut Ruby language and especially those features that create the power of Ruby and distinguish it from other programming languages.

### 4.1.1 Basic information

#### History

Ruby language has been created by Yukihiro Matsumoto also known as "Matz" for english speaking programmers. It firstly appeared publicly in 1995 with version 0.95 in order to reach version 1.0 a year later. As of the time of writing this document, the latest release of Ruby is of branch 1.9 (specifically 1.9.2), but the branch 2.0 with brand new exciting features is emerging on the horizon.

While creating Ruby Matz stated that he focused on developing a language that would be programmer-friendly and that would make the barrier between idea and putting this idea in code as low as possible. The most famous quote of Matz fully describes the philosophy of the language:

> Ruby is designed to make programmers happy

#### Implementation

The official implementation of Ruby has been written in C language. As no particular standard has been developed for the language, the official implementation is a reference for all other vendors. Nonetheless, there is plenty of other implementations including one operating in JVM(JRuby[1]), .NET framework(IronRuby[2]) and Objective-C runtime(MacRuby[3]).

### 4.1.2 Features

#### Basic features

It is worth mentioning few basic features of Ruby language that would make the further part of this chapter clearer.

- Ruby is scripting language - statements are executed as provided and there is no special `main` function
- Ruby is dynamically typed
- Method invocations do not need to include parentheses

#### Everything is an object

Ruby is an object oriented language. In fact, every value in Ruby is a object. Even such "primitive" values as integers, true, false or nil (which is Ruby's NULL). Thanks to that, Ruby promotes and enforce object oriented programming.

Moreover, you can have methods on integers or other primitive values which makes the code much more readable, shorter and enjoyable. For example instead of having Java-like:

```
NumbersConverter.arabicToRoman(1);
```

You can have much more readable:

```
1.to_roman
```

Ruby standard library does not include `to_roman` function for integers, but we could easily add one by using next described feature.

[1] http://www.jruby.org/
[2] http://ironruby.codeplex.com/
[3] http://www.macruby.org/

### All classes are open

Ruby gives us freedom to change already declared classes. Moreover, it does not mean that we can change only the classes that we defined. We can change even classes defined by Ruby standard library. Therefore we can open for example class Integer that represents all integer values in Ruby code and add method `to_roman`:

```
class Integer
  def to_roman
    # Code that would change
    # arabic number to roman one
    # and return it as string
  end
end
```

Now, we would be able to have following line in our code:

```
4.to_roman #returns "IV"
```

### Power of blocks

This is one of the most exciting features of Ruby language. Block is a fragment of code that can be passed to a function and the function may call this code anywhere inside its body.

Block in ruby can be denoted in two ways:

```
do
  #here code
end

#or

{
  #here code
}
```

Usually the first way is used for blocks that span throughout multiple lines, while the second way is used for single line blocks.

Blocks can be used in number of ways. The most popular are iterators:

```
a = [1, 2, 3]
a.each do |i|
  puts i
end
```

Above code would result in printing 1, 2, 3 to the output. Statement `do |i|` means that this block expects one argument called `i` just like regular functions do.

In order to understand how blocks operate, let me present simple example:

```
class Array
  def each_nested(&block)
    for i in 0...self.size do
      if self[i].is_a? Array
        self[i].each_nested &block
      else
        yield self[i]
      end
    end
  end
end

multidimensional_array = [
  [11, 12, 13],
  [21, 22],
  [31, 32, 33, 34]
]

multidimensional_array.each_nested { |element| puts ↩
   element }

# Execution Result:
# 11
# 12
# 13
# 21
# 22
# 31
# 32
# 33
```

```
# 34
```

The above example adds the `each_nested` method to an array that allows us to perform operations on every element of multidimensional matrix. It iterates over every element of an array. If given element is again an array then it recursively calls `each_nested` on it and if it is other element then it passes the control to a block along with the element as a parameter.

As we can see appropriate use of blocks may be very useful and can make code much shorter and more readable

### 4.1.3 Summary

This chapter demonstrated basic features of Ruby language. In order to get more information on this language I forward the reader to the official website[4] or to some books from bibliography[4].

## 4.2 RubyGems

### 4.2.1 Basic Information

RubyGems[5] is the packaging system to distribute libraries for Ruby language. The packages and libraries are called gems and console front-end script is called (not surprisingly) `gem` (distributed along with ruby 1.9, for ruby 1.8 needs to be installed separately). With the help of RubyGems Ruby program can be easily enhanced our with variety of functionalities (for instance serializing/deserializing JSON, handling weekends and holidays in Dates, authentication system etc).

In order to use particular gem first it has to be installed with `gem` command

```
gem install some_gem
```

And then it has to be added to application by first requiring rubygems itself and then requiring the gem.

```
require 'rubygems'
require 'some_gem'
```

[4]http://ruby-lang.org/
[5]http://rubygems.org/

With those few lines of code it is possible to add really extensive features to Ruby programs, as the community gets bigger and bigger and, moreover, its members are willing to share the code.

### 4.2.2 Dependency management

Of course, every gem can use other gems in order to build up on their functionalities. However, in order for those gem to function properly all their dependencies. Moreover, different version of the same gem can differ not only in the implementation, but also in their API. Therefore gems need to keep track of their versions. Surely, it would be totally inefficient if users would have to take care of this on their own. This is why RubyGems take care of it for us.

Every gem is described by its gemspec (gem specification) file. Gemspec is simply Ruby code that defines particular gem. Below is presented gemspec for Ruby on Rails:

```
version = File.read(
    File.expand_path(
      "../RAILS_VERSION",__FILE__)).strip

Gem::Specification.new do |s|
  s.platform    = Gem::Platform::RUBY
  s.name        = 'rails'
  s.version     = version
  s.summary     = 'Full-stack web application framework.'
  s.description = 'Ruby on Rails is a (...).'

  s.required_ruby_version     = '>= 1.8.7'
  s.required_rubygems_version = ">= 1.3.6"

  s.author            = 'David Heinemeier Hansson'
  s.email             = 'david@loudthinking.com'
  s.homepage          = 'http://www.rubyonrails.org'
  s.rubyforge_project = 'rails'

  s.bindir             = 'bin'
  s.executables        = ['rails']
  s.default_executable = 'rails'
```

```
  s.add_dependency('activesupport',  version)
  s.add_dependency('actionpack',     version)
  s.add_dependency('activerecord',   version)
  s.add_dependency('activeresource', version)
  s.add_dependency('actionmailer',   version)
  s.add_dependency('railties',       version)
  s.add_dependency('bundler',        '~> 1.0')
end
```

As seen above, gemspec is pretty straightforward. Along with other attributes, developer can specify dependencies of his gem. Then, when user installs it RubyGems traverses the list of dependencies and installs all of them (of course in the meantime traversing their gemspecs and installing theirs dependencies, which results in the entire tree of dependency). That way, RubyGems becomes really efficient tool to handle libraries.

## 4.3 Ruby on Rails

### 4.3.1 Introduction

Ruby on Rails is a comprehensive framework for web applications development. It has been created by David Heinemeier Hansson in 2004 and a year later he earned Google and O'Reilly's Hacker of the Year award. Its main characteristic is promoting convention over configuration, what enables developers to quickly build even complex web apps without the need to write lengthy XML configuration files. It utilizes Model-View-Controller pattern, which happens to be very good choice for web applications and helps produce clean and maintainable code.

Ruby on Rails is distributed as a gem for Ruby language. After just one command (provided that Ruby and RubyGems are installed on the system):

```
gem install rails
```

the user has entire environment for creating web applications (along with development server).

Among others, Ruby on Rails has two main parts:

**ActiveRecord** Object-relational mapper

**ActionPack** Framework for handling requests and rendering responses

This two parts mainly create the power of the framework and will be described in more details later.

#### Structure of the application

As mentioned before, Ruby on Rails takes its power and agility from "convention over configuration" rule. Therefore, Ruby on Rails applications have very well defined directory structure. The user can easily generate one for his application by using following command:

```
rails new name_of_application
```

Or in case the user is using older (below 3.0) version of Rails:

```
rails name_of_application
```

This command generates the entire directory tree that is used in Rails application. Most important directories are described below.

**app** Contains main code of applications. It is divided into models, controllers and views subfolders

**config** Holds configuration files

**db** Holds files that regard database

**lib** Contains files that create environment for the application but are considered as strictly connected to business logic of application

**public** Place for assets of the application: static pages, stylesheets, javascript files - anything that can be statically served.

Thanks to such well-defined structure, Ruby on Rails knows exactly where to look for particular parts of the system. Moreover, it does not need to go through all the files, but rather load them on demand.

### 4.3.2 ActiveRecord

#### Introduction

ActiveRecord is one of Ruby's Relational Object Mappers, but has been developed as part of Ruby on Rails framework. As whole framework, ActiveRecord philosophy is convention over configuration.

Usually, models in Rails applications are stored in app/models directory. They are classes that derive from ActiveRecord::Base class. And, in fact, this is enough to handle simple models. It is possible to create persistent business models with great ease.

```
class Item < ActiveRecord::Base

end
```

Above example depicts the most simple model. Of course, the user needs to set up configuration for database connection. Usually it is done in config/database.yml file while in Ruby on Rails project or provide paramaters directly to ActiveRecord (if not used in Rails project).

```
ActiveRecord::Base.establish_connection(
  :adapter  => "postgresql",
  :host     => "localhost",
  :username => "database_user",
  :password => "passme",
  :database => "my_database"
)
```

### Basic Model

Coming back to the code example of model:

```
class Item < ActiveRecord::Base

end
```

This will map automatically to items table in the database. It is not necessary to specify what kind of columns, the table has. All attributes of the model are automatically drawn from the table and (thanks to dynamic nature of ruby) translated into appropriate accessor methods for Ruby class.

So assuming the table has been created with following sql statement

```
class Item < ActiveRecord::Base

end
```

Then with the ruby code presented above we get following functionallity:

```
item = Item.new
item.name = "Item 1"  #=> "Item 1"
item.quantity = 5     #=> 5
item.save             #=> true
# The item got persisted to database
# Now it is possible to retrive it:
item2 = Item.find(:first)
#=> <Item id:1, name: "Item 1", quantity: 5>
item2.name      #=> "Item 1"
item2.quantity  #=> 5
```

Moreover, it is possible to search for records by forming (even very complex) queries:

```
Item.where(:quantity => "5").order("name DESC")
```

### Associations

As every Object Relational Mapper, ActiveRecord enables developers to specify relations between models. The only thing that needs to be done in order to handle associations well is add the foreign key column to the database and describe the relation in the model by using one of three class methods:

- has_many
- has_one
- belongs_to

This is shown in the example:

```
# CREATE TABLE categories (
#   id SERIAL PRIMARY KEY,
#   name VARCHAR(255)
# );
#
# CREATE TABLE items (
#   id SERIAL PRIMARY KEY,
#   name VARCHAR(255),
#   quantity INTEGER,
#   category_id INTEGER REFERENCES categories(id)
# );
```

```
class Item < ActiveRecord::Base
  belongs_to :category
end

class Category < ActiveRecord::Base
  has_many :items
end

# Application code
category = Category.new(:name => "Category 1")
category.save
item = Item.new(:name => "My Item")
item.category = category
item.save
```

### Validations

As models usually keep great deal of application's business logic there is another feature that is very useful - validations. It is possible to define rules against which the model will be tested. If some of those conditions would not be fulfilled, the record would not be saved and ActiveRecord would store Errors object that gathers information on unsatisfied validations.

```
class Item < ActiveRecord::Base
  validates_presence_of :name
  validates_uniquness_of :name
  validates_presence_of :quantity
end

#Application code
item = Item.new(:name => "Item 1")
item.save      #=> false
item.errors    #=> {:quantity=>["can't be blank"]}
item.quantity = 4
item.save      #=> true
item.errors    #=> {}
```

```
item2 = Item.new(:name => "Item 1", :quantity => 5)
item2.save     #=> false
item2.errors   #=> {:name=>["has already been taken"]}
item2.name = "Item 2"
item2.save     #=> true
```

Functionality presented above is just a small subset of possibilities. It is possible to specify when given validations should be executed (during create or update of the record) or specify conditions that turn off validation (eg. do not check presence of one attribute if the second one is true). Moreover, except for standard `validates_*` functions, the user can specify his own validations by passing the names of methods that would be executed during validations to `validate` class method.

### Migrations

Among others, ActiveRecord provides another feature that makes the life of developers just a little less painful - Migrations. They are used to handle changes to the database, and as an extension - to structure of Models.

So, instead of creating and altering tables from SQL we could use a migration that looks like this:

```
class AddItems < ActiveRecord::Migration
  def self.up
    create_table :items do |t|
      t.string  :name
      t.integer :quantity
    end
  end

  def self.down
    drop_table :items
  end
end
```

Rails provide tools that make work with migrations like generators that help create migration files or tools for running migrations (`rake db:migrate`). Moreover, in the applicaiton database Rails would create a table called `schema_migrations` that keeps track of which migrations have been applied to given database. This is useful, when working on some project with fellow developers - if one of them adds a migration and

then the other developer pulls it from the version control system, he can run `rake db:migrate` and Rails would run only new migrations.

#### Summary

As stated in the introduction, it is clear to observe, that ActiveRecord provides great deal of features that surely makes the rapid development of applications a breeze. Moreover, thanks to Ruby's nature it all is packed in very easy and accessible API.

The intent of this chapter was to make a short introduction to ActiveRecord and it did not cover all of interesting and useful features of ActiveRecord. There is plenty of other things such as callbacks, observers, named_scopes that are worth noting, but in order to keep this document as focused on the topic as possible they are omitted here. In order to get more information, the reader can refer to ActiveRecord documentation.

### 4.3.3 ActionPack

#### Introduction

As said before, Ruby on Rails is build on the Model-View-Controller pattern. Model part is handled by ActiveRecord described in previous chapter. ActionPack takes care of two remaining parts - View and Controller. So, in general ActionPack is divided into two parts:

- ActionController
- ActionView

It is easy to guess which part is responsible for which part of MVC pattern.

#### From request to response

In very short, when Ruby on Rails application gets a request, first, it parses it in order to translate the request into form that is better to use programatically. Then, based on the routing rules set in your application Rails determines which action from which controller to call. Then it creates an instance of particular controller and calls method for target action and passes the parameters that came along with the request.

Then the particular controller takes over. Its responsibility is to understand the data provided with request, conduct some actions based on them and render appropriate

response. The response is usually rendered by using some template that is filled with data provided by the controller.

### ActionController

Controllers in Ruby on Rails are just subclasses of ApplicationController, which inherits from ActionController::Base. Actions are defined by just simply adding instance methods to controller class.

The most basic controller for blog posts could look like this:

```
class PostsController < ActionController::Base
  def index
    @posts = Post.all
  end

  def new
    @post = Post.new
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      redirect_to post_path(@post)
    else
      render :new
    end
  end

  def edit
    @post = Post.find(params[:id])
  end

  def update
    @post = Post.find(params[:id])
    if @post.update_attributes(params[:post])
      redirect_to post_path(@post)
    else
      render :edit
```

```
    end
  end
end
```

The controller presented above has 5 basic CRUD actions. It can be observed, that controller assigns instance variables - this is in order to be able to read those variable in the view. Moreover, some of the actions of this controller do not have `render` command which would cause the chosen template to be processed. This is correct, because Rails by default (if no `render` has been specified) looks for template in default path - in this case it would be `app/views/posts/action_name`, so for example for `index` action it would look for `app/views/posts/index.html.erb`

### ActionView

ActionView is Ruby on Rails' way to generate html (and not only - xml and others can also be produced) views. ActionView by default provides three ways to generate views:

**erb** ERB (Embedded Ruby) templates - mix of html and embedded ruby code

**builder** builder for XML documents

**rjs** generator of javascript actions

The most common are ERB templates. They enable developers to easily define HTML views. The example of ERB template could look like this:

```
<html>
  <head>
    <title>
      <%= @post.title %>
    </title>
  </head>
  <body>
    <h1>
      <%= @post.title %>
    </h1>
    <p>
      <%= @post.body %>
    </p>
    <p class="published_at">
```

```
      <%= @post.published_at %>
    </p>
  </body>
</html>
```

As said in previous chapter, the instance variables of controller are available in the view. That is why it is possible to use `@post` variable and thanks to this it is possible to extract all the logic into controller and have clean views.

**Summary**

This chapter was just the introduction into ActionPack in order to give an overview of how it works. It does not cover more advanced features as filters in controllers or layouts and partials in views.

## 4.4 Metaprogramming

Metaprogramming is programming of code that its input and output is other code. It is used to manipulate the behavior of the program during runtime depending on other parts of the application.

Thanks to dynamic nature of Ruby it provides comprehensive reflection API. It allows developer to modify classes, list or add methods to classes or even particular objects.

### 4.4.1 Metaprogramming in Ruby

The way Ruby language is constructed makes it really easy to apply metaprogramming to. First of all, it is a dynamic language which means that things like adding new code or extending objects can be done during run time. It is kind of a direct result of the fact that Ruby is an interpreted language. Anyway, this single characteristic makes it very useful in terms of metaprogramming. What is more, as everything in Ruby is an object so are classes (they are instances of class Class). Therefore, we can act on them as we would do on objects, what sounds like even bigger improvement on metaprogramming.

As said before, developer can define methods during runtime. Moreover, it is possible to reopen class definitions multiple times and add methods. The following example depicts this

```
class MyClass
end

object = MyClass.new

puts "a or b?"
choice = gets.chomp

case choice
when "a"
  class MyClass
    def what?
      puts "This is 'A'"
    end
  end
when "b"
  class MyClass
    def what?
      puts "This is 'B'"
    end
  end
else
  class MyClass
    def what?
      puts "Unknown"
    end
  end
end

object.what?
```

The class `MyClass` is first created, but without any methods. Then the object of this class is created. So far it is pretty standard, but next the user is asked for an input. Then depending on this input the class `MyClass` is reopened and method `what?` is defined with implementation depending on user input. Then the method is called on previously created object. After running this program the user would be able to observe that depending on what is his input the appropriate method definition is applied.

The example above shows the dynamic nature of Ruby. But this is only the beginning. Thanks to so-called Singleton Classes in Ruby it is possible to define methods on particular objects. So it is possible to rewrite the example in such way:

```
class MyClass
end

object = MyClass.new

puts "a or b?"
choice = gets.chomp

case choice
when "a"
  def object.what?
    puts "This is 'A'"
  end
when "b"
  def object.what?
    puts "This is 'B'"
  end
else
  def object.what?
    puts "Unknown"
  end
end

object.what?
```

This time `what?` method has been defined for `object`. If developer would create another instance of class `MyClass` it would not have `what?` method. As soon as method has been defined for particular object the class of this object has been changed to singleton class - a subclass of `MyClass` - and it keeps all the methods defined for the `object`. If digged deeper into Ruby internals it would be possible to see that such class has "singleton" flag set to true. This is an indication for Ruby that this class is only a helper class and would not be seen in example in `object.class` call (this method would still yield `MyClass`).

Those examples show the dynamic nature of Ruby language. In addition to features presented above, Ruby provides a lot more. For example, it gives the ability of evaluating a string as Ruby code. This functionality is provided by following methods:

- eval
- instance_eval
- class_eval

The difference between those methods is the context the string is going to be evaluated in. `Eval` evaluates code in current context (or in the context passed to eval by providing Binding object as second argument), `instance_eval` is called on some object and evaluates the code in context of the receiver of the method while `class_eval` can be called on a Class or Module and evaluates the code in context of given Class/Module.

Moreover, the developer can find more methods that make the reflection API complete:

- methods
- protected_methods
- private_methods
- define_method
- remove_method
- undef_method

Names of the methods are self explanatory. The difference between `remove_method` and `undef_method` is that `remove_method` removes implementation from particular module but Ruby will still traverse the inheritance tree to find implementations of given method in parent classes. `Undef_method`,on the other hand, would prevent the module from responding to given method at all.

### 4.4.2 Metaprogramming in Rails

As Rails is a framework for Ruby it is sane to think that it would provide the same level of dynamism and same level of metaprogramming capabilities. And this assumption turns out to be valid. First of all, developers can make use of all Ruby metaprogramming capabilities. But Ruby on Rails introduces some new capabilities on top of Ruby plus adds functionality that is specific for web apps, object relational mapping etc.

One of most useful methods added in Rails is constantize methods. It takes a string as an argument and returns a constant that is found in the environment under this name.

Therefore, as Ruby classes are just constants that point to particular objects of Class we may get the classes by their names in string as presented in the example:

```
require 'rubygems'
require 'rails'

class Dog
  def self.voice
    puts "Whoof"
  end
end

class Cat
  def self.voice
    puts "Meow"
  end
end

class Cow
  def self.voice
    puts "Moo"
  end
end

puts "Type in animal you want to hear"
puts "Choices:"
puts "Dog, Cat, Cow"
choice = gets.chomp
choice.constantize.voice
```

Such usage of constantize may be very useful, especially in kind of application this thesis presents, as it would allow to pass the names of entities around as strings and dynamically retrieve classes in the code.

In addition, ActiveRecord provides comprehensive reflection API. Developer can get the list of all columns the entity has in database by calling `columns` method on the entity class. That method returns list of subclasses of `ActiveRecord::ConnectionAdapters::Column` which provides information on name, type and other metadata of the column.

In order to fully make use of this API and effectively manage entities there is another method that may be helpful. Developer can call `reflect_on_all_associations` which would return the array of objects representing all associations (`has_many`, `belongs_to`, `has_one`) the particular entity has. Moreover, if you pass association type as an argument it would return only associations of this type.

Features shown above would definitely make writing the targeted plugin much easier and enjoyable.

# Chapter 5

# Process and methodologies of solution development

## 5.1 Creating a RubyGem

The first step in creating the plugin is setting up the structure of directories and creating necessary files for this plugin to be available for installation. Chapter 3.2 provided basic information about the RubyGems standard. This chapter will provide some further details on how the structure for particular RubyGem has been created.

### 5.1.1 Structure of directories

The essential structure of directories needed in RubyGem standard is a `gemspec` file in the root directory of the Gem and the `lib/` directory containing Ruby file with the name corresponding to the name of the Gem. So in the case of the Gem described in the application, which would be called Administer, the name of the file would be `lib/administer.rb`. This file is automatically loaded when the Gem is added to the program. You could put all the code in that file, but as the project gets bigger it is common practice to split the code into multiple files and require those files in the main (`administer.rb`) file.

Instead of creating the structure manually, there are plenty of options that would help create the structure automatically. The one to be considered the best recently is using bundler, which is a gem for managing other gems in the application. Bundler among others, provides `bundle gem` command which creates structure needed for the newly

created Gem. Moreover, it initializes `git` repository in the Gem directory for Version Control:

```
$ bundle gem administer
      create  administer/Gemfile
      create  administer/Rakefile
      create  administer/.gitignore
      create  administer/administer.gemspec
      create  administer/lib/administer.rb
      create  administer/lib/administer/version.rb
Initializating git repo in /Users/piotrj/Projects/↩
   testytest/administer
```

### 5.1.2 Gemspec

The `bundle gem` command creates a template for gemspec file:

```
# -*- encoding: utf-8 -*-
$:.push File.expand_path("../lib", __FILE__)
require "administer/version"

Gem::Specification.new do |s|
  s.name        = "administer"
  s.version     = Administer::VERSION
  s.platform    = Gem::Platform::RUBY
  s.authors     = ["TODO: Write your name"]
  s.email       = ["TODO: Write your email address"]
  s.homepage    = "http://rubygems.org/gems/administer"
  s.summary     = %q{TODO: Write a gem summary}
  s.description = %q{TODO: Write a gem description}

  s.rubyforge_project = "administer"

  s.files         = `git ls-files`.split("\n")
  s.test_files    = `git ls-files -- {test,spec,features↩
     }/*`.split("\n")
  s.executables   = `git ls-files -- bin/*`.split("\n").↩
     map{ |f| File.basename(f) }
  s.require_paths = ["lib"]
```

```
end
```

The gemspec file defines the specification of the gem. In addition to the attributes visible above (which are self-explanatory) there are few other that may be pretty important:

**add_dependency** Adds other gem as dependency

**add_development_dependency** Adds ohter gem as dependency for development

### 5.1.3 Gem building

When the gem's code is ready it is necessary to create a package. It can be achieved by running

```
gem build gem_name.gemspec
```

This results in creating `.gem` package, which is a binary file.

### 5.1.4 Gem publishing

It is possible to distribute gem by sending the `.gem` file to other people via email or putting in on ftp server. But, there is unified way of distributing gems which is `rubygems.org` server.

Publishing gem on `rubygems.org` is as easy as running

```
gem push name_of_package.gem
```

where `name_of_package.gem` is a file we got after building the gem. In order to be able to push to rubygems developer has to have an account and will be prompted for login credentials.

## 5.2 Scrum

In order to ensure that the works both on Master Thesis and on the plugin go smoothly and according to plan I decided to use SCRUM methodology to lead the project. As SCRUM is usually intended for multi-person teams, it needed to be slightly adapted for the needs of the project of this Master Thesis. Nonetheless, it helped to evenly distribute workload in time and avoid exhausting bursts of intensive work when some deadline was approaching.

### 5.2.1 What is Scrum?

The idea behind Scrum is to divide the entire project into very small subprojects that can be done in intervals of up to 4 weeks. Such intervals are called `sprints`. The result of sprint is supposed to be part of the project that can be added to the production system. So most likely, it would be a complete feature that can be developed independently of other pending features and added to the existing system, what would accomplish the `sprint`.

Before the project kicks off, the target functionality is discussed and decided. The functionality is then divided into features that would together build the product. Next, the difficulty of every feature is discussed and estimates on needed time and resources are drawn. This step is very important, as all next steps in the project would depend on this initial one.

Each sprint would begin with a short team meeting with a purpose of drawing appropriate number of features from the backlog that would be implemented during that particular sprint. As it can be observed, the estimates from the initial step would become useful at this step as the team knows how much work would approximately it take to finish particular feature. Then the features are prioritized.

During the sprint, developers take the features from the top of sprint's backlog and try to finish all of them before the end of the sprint. As the features has been selected to fit the workforce of the team for particular sprint, this should be doable without overtimes and stressful situations.

Moreover, some implementations of Scrum introduce very short daily meetings, a purpose of which is to keep everyone in the team informed about the status of the work done as well as discussing problems that appeared.

The sprint ends with a team meeting that reviews the performance, what has been done and what could not be completed. In addition, problems are discussed and conclusions are drawn for the future sprints.

### 5.2.2 Scrum for Master Thesis

As the Master Thesis should be and usually is a single person effort not all parts of Scrum can be implemented for the purpose of creating one. Nonetheless, the idea of dividing the whole project into very small and short steps helped me to create the plan of work. In order to embrace the power of meetings as motivational tool I contacted one of Ruby on Rails companies located in Łódź in order to organise weekly meeting

with Ruby on Rails mentor. He would weekly review my work, help me with solving programming problems and guide into the world of Ruby programming best practices. In addition, about every month I tried to contact my supervisor in order to share and discuss my progress.

As for planning work for my weekly sprints, I used Pivotal Tracker[1] to keep track of all tasks that needed to be done in order to get closer to finishing the project. In the begining, when starting the Master Thesis, I decided how the programming project should look like and what would be the parts of the report I would hand in and I would sort all of those issues according to what should be done first. That would create my backlog. Then every week I dragged appropriate number of tasks to the "current" section and at the end of the week I would get asked about.

Not surprisingly, such arrangement helped to keep me motivated throughout the entire process and helped me avoid stress of deadlines and the feeling of not knowing where to go next.

## 5.3 Testing

### 5.3.1 Introduction

The community gathered around Ruby language and especially framework Ruby on Rails is strongly focused on creating software using test oriented methodologies. There are two results of such positive peer pressure. First, most of applications and libraries that appear in the Ruby ecosystem come up with set of comprehensive test suite that can instantly confirm correctness of the application. Second, there are a lot of tools that developers can choose from in order to best satisfy their taste and needs.

### 5.3.2 Advantages of automatic tests

Software can be tested either manually or automatically. Manual testing seems to be a good choice at first as it seems you can test how the application interacts with real human and how it reacts to his input. However, after longer consideration, few drawbacks of such method can be pointed out:

- If developers are used as testers they waste their time clicking through the application while they could could have been doing productive work on the code.

[1] http://pivotaltracker.com

- If someone else is testing the project he needs firstly to spend a lot of time on getting to know the project assumptions and aims.
- It is really hard to manually test every possible input and situation for every new feature.
- It does not give possibility of testing the details of implementations.
- Developers cannot get instant feedback

Moreover, those disadvantages rise along with the growth of the project.

That is why more and more projects introduces software that automatically tests the application and that way diminishes the need of manual testing. One of advantages of this is the fact that not only it is possible to test the end-to-end operation of the application but also test the internal implementation by using `unit tests` for testing individual classes or methods and `integration tests` which test the integration between components of the application.

Automatic testing provides several advantages:

- Ability to run the test suite on every build of application
- Instant notice when some part of the application (even remote from developer is currently working on) fails
- Ability to test components of application in isolation
- Possibility of setting up appropriate state of application environment in order to test every possible situation

### 5.3.3 Test Driven Development

Test Driven Development (also abbreviated as TDD) is one of test oriented methodologies of software development. Its philosophy is summarized as `Red-Green-Refactor-Repeat`, which means:

**Red** Write a test for new feature that should not pass yet (usually denoted by red color)

**Green** Write the most simple code that would make the test pass (denoted by green color)

**Refactor** Rewrite the code so it is well-designed and easy to maintain while still keeping the test passing

**Repeat** Repeat those steps for next feature

One of most important rules of TDD is that the test should be written before any code. That way, the developer has to first understand the feature he needs to add and then write the code. Such approach results in higher productivity.

Test Driven Development suite usually consists of two types of tests: unit tests and integration tests.

### Unit Tests

Unit tests focus on testin one of components of code in isolation. This means that all components that communicate with the component which is a subject of test, would get substituted by pseudo-implementations that give predictable results. So for example for code like this:

```
class A
  def method_a
    b = B.new
    b.method_b
  end
end

class B
  def method_b
    #some code
  end
end
```

If the developer would be testing `method_a` from class `A` we would substitute implementation of `method_b` from class `B` with implementation that gives a result the developer expects. Such action is called mocking or stubbing. That way, the behavior of `method_a` is tested under known circumstances as it is not important at the moment how `method_b`. In one of Ruby testing frameworks RSpec the test for `method_a` could look like that:

```
describe A do
  describe :method_a do
    it "should return the value provided by method_b" do
      b = mock("B")
      b.stub(:method_b => "5")
```

```
      B.stub(:new => b)
      a.method_a.should == "5"
    end
  end
end
```

Thanks to such approach if test fails it is instantly clear which part of the system behaves in a wrong way without the need for lengthy debugging.

#### Integration Tests

The reason of having integration test is to see whether the components in fact cooperate together as supposed. As unit test focus on testing in isolation, some messages that come from other parts of the system may be not taken into account by the developer so it is important to test the whole system to see whether everything is fine.

### 5.3.4 Behavior Driven Development

Another test oriented methodology is Behavior Driven Development (abbreviated as BDD). The idea behind this is to first define the behavior of the application for given feature and then write the code needed for that feature to be working. It sounds a lot like TDD, but the main difference is that the test is written for the interaction with end user - it does not take into account details of implementation, but rather how the application behaves and interacts via user interface. Moreover, those tests are usually written in language which is close to natural. This is caused by the fact, that Behavior Driven Development aims to drag non-technical part of the team into process of testing the application. By defining tests in language that may be understood by anyone we enable managers and product owners to write test cases or reading them in order to verify that the test case in fact satisfies the business need.

Below example of test written for Cucumber (which is one of Ruby's BDD tool) shows how such tests can be written:

```
Feature: Managing posts
  In order to manage my blog
  As a blog owner
  I want to be able to manage posts

  Scenario: Adding new blog post
```

```
Given no posts exist
When I am on the home page
And I follow "new post"
And I fill in "Title" with "Post number 1"
And I fill in "Body" with "Hello world"
And I press "Create Post"
Then I should be on posts list
And I should see "Post number 1"
And I should see "Hello world"
```

### 5.3.5 Tools available for Ruby on Rails

As said before the test-oriented community produced great amount of libraries, frameworks and tools for testing. The most important ones are:

**Test::Unit** Default test framework for Ruby 1.8

**MiniTest** Successor of Test::Unit, default for Ruby 1.9

**RSpec** Test framework that aims to substitute Test::Unit or MiniTest. It provides more BDD-like syntax and has an incorporated mocking framework.

**Cucumber** BDD tool that enables developers to specify their features using Gherkin[2] language.

**FactoryGirl** Tool that aims to substitute fixtures which are used by default in test frameworks.

**Capybara, Webrat** Tools for simulating user interaction with application via web browser

**Steak** Integrates Rspec and Capybara and makes it possible to write end-to-end tests in pure Rspec instead of Cucumber

### 5.3.6 Summary

After considering the ideas and tools I have decided to use a mixture of TDD and BDD in order to grow my application. All new features would be first described in terms of BDD in order to understand exactly how the application should act. Then, while writing code unit test would be written in order to understand what is expected

[2]https://github.com/cucumber/cucumber/wiki/Gherkin

from the implementation and in order to make the design modular and testable which leads to code that is easier to read and maintain.

The tools that will be used are following: Rspec, Capybara, FactoryGirl and Capybara.

# Chapter 6

# Design and implementation

The aim of this chapter is to provide description of how the plugin codebase, show how it has been designed and what decisions has been made. In addition it aims to indicate interesting examples of usage of Ruby and Ruby on Rails API.

## 6.1 Overview

The plugin has been developed as Rails Engine (details will be covered in next chapter). This means that it is a Rails application that can be embedded into another one. The fact, that the plugin is a Rails application provides the ability of using whole power of Ruby on Rails. So, it is possible to use the MVC pattern, set up routing for the application and so on.

In order to get the plugin running following parts had to be developed:

- Setting the application as `Rails Engine` as said before.
- Handling the `Models` - finding all models in the application as well as getting the model class from string representation
- Getting the list of `Fields` in the model along with their type and any necessary attributes that would be necessary for generating the form for given model. This includes not only fields such as text or integer fields, but also handling associations between models which is the most challenging task.
- Elaborate a convenient way of defining `Configuration` for the plugin that would be both flexible and easy to use at the same time.

- Develop the web part of the plugin - setting up the `Routing, Controllers and Html Templates` that would allow users to smoothly use the application.

The plugin has been developed along and tested with a simple application that would act like a simple blogging system. Appropriate features such as particular types of model associations has been added to the test application when needed in order to see whether the plugin handles it in correct way. As test application is not integral part of the plugin it would not be described in details. In short, the models that would appear in the test application are following:

**Posts** used as main model to test all associations on. It has standard fields such as string field for title, text field(long string) for body, date field for published at date.

**Categories** can be attached to Post in association one-to-many. Used to test belongs_to association in Post.

**Comments** attached to Post in many-to-one association. Used to test has_many association.

**Attachment** attached to Post in one-to-one association. Used to test has_one association.

Association testing has been shown from the Post model perspective, but the plugin would handle the associations from both ends as it needs to be universal and be working for every application.

## 6.2 Rails Engine

In order to create a web application that would be mountable into other Rails applications it has to be declared as an Engine. It can be obtained by defining an Engine class in the gem module:

```
module Administer
  class Engine < Rails::Engine
  end
end
```

This few lines of code makes the gem a Rails Engine. This means that it will load the code located in `app/` and `config/` directories. This is acomplished by the single fact of creating a class that inherits from `Rails::Engine` as the internals of that

class set up the load path for those directories. In fact, from version 3 of Rails every `Rails::Application` is `Rails::Engine`.

Moreover, `Rails::Engine` or in fact its superclass `Rails::Railtie` enables developer to hook in into Rails initialization process by adding initializing routines to the list. This would become useful soon when the plugin would need to do some actions after all Rails initialization has been done (will be covered in Fields chapter).

## 6.3 Models

A `Model` class is a central point of the administer plugin. It is responsible for handling models, getting the list of available models and so on.

### 6.3.1 Getting list of all models

First thing that needed to be done in respect of models was to find all models in the application. There are few possibilities of obtaining the list of classes in current application. The first that has been tested was getting all subclasses of `ActiveRecord::Base`, which is a standard superclass for models in `ActiveRecord`:

```
module Administer
  class Model
    class << self
      def all
        ActiveRecord::Base.subclasses
      end
    end
  end
end
```

This works fine, but there is one thing that is not necessarily proper - hardcoding. It hardcodes the use of `ActiveRecord`. And, as the plugin should be as universal as possible, it is not a good idea to narrow the possibility of extending into other ORMs this early in implementation. Moreover, not all ORMs use a superclass as creating models. Some of them like `DataMapper`[1] or `Mongoid`[2] use a mechanism of including a module in a class so instead of:

[1] http://datamapper.org/
[2] http://mongoid.org/

```
class TheModel < ActiveRecord::Base
end
```

it becomes:

```
class TheModel
  include DataMapper::Resource
end
```

So there is a need for some other solution. Fortunately, Rails has a predefined structure of directories and it is known where the models (usually) are kept - in `app/models` directory. Therefore, such an implementation has been derived:

```
def all
  if @@models.empty?
    Dir[Rails.root.join('app/models/**/*.rb')].each do |↩
       path|
      model_name = path.split('/').last.gsub(/\.rb$/, '')
      @@models << Administer::Model.for(model_name)
    end
  end
  @@models
end
```

where `Model.for` is a method that returns a class with a name represented by given string. What this implementation does is - it looks for all `.rb` files in models directory and figures out models by looking at their names. This solution seems better as we no longer hardcode ActiveRecord. Howevwer, one more assumption here is done - usage of `app/models` as a directory for models. Since it is configurable in Rails it would be nice to handle the situations when the developer defined the models' directory to be something else. Fortunately, it is possible to get the configuration out of Rails. It can be obtained like that:

```
def all
  if @@models.empty?
    Rails.application.paths.app.models.paths.each do |↩
       models_path|
      Dir[File.expand_path("**/*.rb", models_path)].each ↩
         do |path|
        model_name = path.split('/').last.gsub(/\.rb$/, ↩
           '')
```

```
        @@models << Administer::Model.for(model_name)
      end
    end
  end
  @@models
end
```

Above implementation takes into account all directories defined in Rails as containers of models files by using `Rails.application.paths.app.models.paths.each`

### 6.3.2 Getting the class from string

As all parameters of HTTP request are in fact `Strings` there has to be a routine for changing a `String` into a `Class`. It is needed, because the user would in HTTP request define which resource he wants to access. One way would be to keep a hash where keys would be names of classes in string form and values would be classes itself. But, since Ruby is a dynamic language with comprehensive reflection API that has been enriched with few methods by Rails it is possible to get it much smarter and have the implementation like this:

```
def lookup(model_name)
  model_name.camelize.constantize
end
```

It takes a string, camelizes it - which means `"my_model"` would become `"MyModel"` and then calls `constantize` on that string. This method (as described in 3.4.2) takes a string and returns a constant with that name.

Then, in order to make the API flexible there is method `Model.for` defined that returns a `Model` instance for particular class

```
def for(model_name)
  klass = model_name.is_a?(Class) ? model_name : lookup(↩
     model_name)
  Model.new(klass)
end
```

Therefore it can be called either `Model.for("my_model")` or `Model.for("MyModel")` or even `Model.for(MyModel)` if needed depending on a context.

## 6.4 Fields

Proper handling of fields of the model is probably the most tricky part of the application. If it was only for the fields in the database then it would not be that complex, but as the plugin is supposed to handle associations between models as well - that introduces the challenge.

### 6.4.1 Representation of fields in the system

At first there was an idea of passing around the list of fields for given model as the list of hashes. The hash for field would keep all information important for given type, so the most basic would keep:

- Name
- Type

and if needed it could keep other info, for example:

- Foreign Key
- Association Class
- Parent

The example of list of fields could look like:

```
[{:name => :title, :type => :text_field},
 {:name => :body, :type => :text_area},
 {:name => :publish_at, :type => :date},
 {:name => :category, :type => :belongs_to, :↩
    association_class => :category, :foreign_key => :↩
    category_id}]
```

However, such approach is not very flexible. If new type of field needs to be added the code for handling it should be changed in every place, where that list of hashes is accessed. Moreover, handling such list of hashes needs a lot of messy code itself, with a lot of conditions.

Therefore, there emerged the need for some more comprehensive representation of the fields. More object oriented approach has been introduced. Every type of field would be represented by the class that would extend the most basic `Administer::Fields::Base`, which would provide the most fundamental constructor and set of methods. The hierarchy is presented in figure 6.1 (page 61)

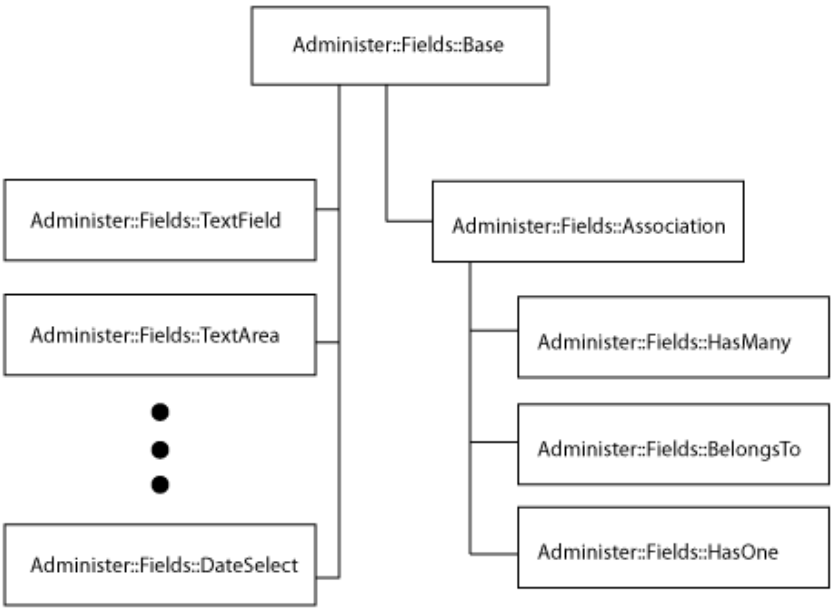



Figure 6.1: Diagram of structure of classes representing fields

### 6.4.2 Mapping of actual fields into their representation

As there is already the structure that will represent the fields in the system, the only problem was to map the actual fields and associations into what they should be. This process consists of two parts:

- Getting the list of fields and associations from the model
- Mapping to appropriate classes

#### Getting the list of fields and associations from the model

ActiveRecord reflection API enables developer to get the list of all database fields for given model by using `columns` method. This returns the list of `ActiveRecord::ConnectionAdapters` subclasess (depending on database adapter it would be `SQLiteColumn`, `PostgreSQLColumn` and so on). The `ActiveRecord::ConnectionAdapters::Column` class contains all kind of data for the column, but the most important from the perspective of the plugin is `name` and `type`.

The list contains all columns, which means that the foreign keys for `belongs_to` associations are there as well. They would have to be substituted by the associations themselves.

Moreover, in order to get the associations the `reflect_on_all_associations` method can be used which returns the list of all association reflections objects. They are rep-

resented by `ActiveRecord::Reflection::AssociationReflection`, which contains information on the type of association, class of associated objects etc.

After combining those lists together and replacing `belongs_to` foreign keys columns with association representations the list is ready to be transformed into our representation of fields.

### Mapping fields to appropriate representation classes

Next step is to map obtained objects into objects that would be useful for the purpose of rendering form or displaying the list of records. For this purpose the `FieldBuilder` class has been introuced. Its responsibility is to keep rules for such mapping and perform the transition for given set of fields.

Again, flexibility has been an issue here. In order to not hardcode the rules in the class, the interface for registering rules has been developed. The `FieldBuilder` class introduces `register_class` method which takes two parameters - the class and the block that describes a mapping rule for given class.

For instance, the definition of (shortened) rule for `Column` from `ActiveRecord` may be the following.

```
FieldBuilder.register_class ActiveRecord::↩
   ConnectionAdapters::Column do |column|
  case column.type
  when :text
    TextArea.new(column.name)
  when :date
    DateSelect.new(column.name)
  else
    TextField.new(column.name)
  end
end
```

Those rules are kept in a special type of hash (from gem called SuperclassHash[3], which has been developed for the purpose of that plugin as well). The keys of this hash are classes. Then, when program asks for the value for some key(class) and value has not been found, the superclasses are tested up to the point where value is defined or `Object` class is reached which is the parent of all classes and objects in ruby.

[3]https://github.com/piotrj/superclass_hash

When `FieldBuilder` gets the list of field objects it iterates over the list and for every object finds appropriate rule and calls it with the object as argument.

One of problems that appeared was the fact, that while defining the rule above, the environment has to have all the classes for `ActiveRecord` initialized. The connection adapters are initialized at the very end of Rails initialization process. So the rules definition could not have been loaded along with the plugin as it is loaded in the beginning of the process.

But, as said before in the chapter describing Rails Engine - Ruby on Rails provides an API for hooking into initialization process. Moreover, it is possible to choose the initializer before or after which the routine should be executed. Therefore, it enables delaying the moment of rules registration to the point where all elements required are there:

```
initializer "administer.initiailze_orm", :after => :↩
   finisher_hook do
  require File.expand_path("active_record", File.dirname(↩
     __FILE__))
end
```

This is how the newly created list of fields represented by the classes defined in the beginning is created.

## 6.5 Configuration

### 6.5.1 Need for configuration

Although, the plugin is supposed to be "plug and play" there are some situations that cannot be handled without any user input. Let's take a look at probable situation that there are two models:

- Category
  - name
- Post
  - title
  - body
  - belongs_to: Category

Let's assume that `belongs_to` associations are represented by select field. The following problem appears - how to display the category in the select field. Should it be just the id of Category (id is the only field that would appear in every single model) ? It would not be very helpful for user as seen on figure 6.2.


Figure 6.2: Select with id

Another idea would be to use `to_s`. But the default one is not very informative either (fig. 6.3). And making developers to redefine `to_s` in every model contradicts the entire idea of the plugin being unobtrusive.


Figure 6.3: Select with to_s

Very similiar situation happens with `inspect` method which in plain Ruby uses `to_s`, but in Rails has been redefined to be a little bit more informative (fig. 6.4). Nevertheless, it is not very user friendly in terms of web interface.

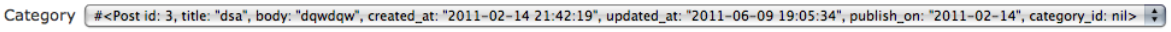



Figure 6.4: Select with inspect

None of the solutions presented above is correct or even remotely appropriate. Moreover, the application is not yet smart enough to tell by itself which fields should be used for display name of given model. For human being it is pretty obvious, but machines are not there yet. It could require every model to have a field or method name that would be used, but again, that would be the violation of unobtrusiveness rule. Therefore, the user needs some way of telling the application, how he wants given model displayed.

In addition, there are much more situations that need some user input - maybe some models of field would be hidden. Or some kind of authentication and authorization should be set up for the administer section. So, there is a strong need for some kind of configuration. It should be easy enough to still keep the feeling of being "plug and play". User should not be forced to write tedious XMLs or spend more time on it than completely necessary.

### 6.5.2 Domain Specific Language

#### The idea

As said before, forcing developers to define some methods with predefined names is not an option. One of the assumptions of the project was not to interfere with the code of application. So it needs some other way of configuration.

Rails application have special place for things like that. There is a directory `config/initializers` that is dedicated to hold files that would be loaded during the initialization process of application. They are usually used to set up configuration for gems, plugins and other libraries. This is the place where the user would hold configuration for administer.

Another concern was the form that the configuration would take. The first thought was to create the most straightforward way - just provide a set of class methods for `Administer::Config` that would allow people to set appropriate values. One rule could look like that:

```
Administer::Config.set_display_name(Category, :name)
```

where `Category` is the model and `:name` is the symbol representing method that should be called.

Looks correct, and probably would be enough for most of use cases. However, soon some developer would probably face the fact that none of existing methods or fields are appropriate to display as name for the model. Some field may need to be trimmed or the best match would be to combine few fields (eg. first and last name of a person). The immediate solution would be to define the method in model that prepares the name and then pass that to configuration. And as long as that method would be defined and used in the application anyway it is all right. But if it is created just for the purpose of using it in configuration it is again kind of situation where some unnecessary code clutters the code of application. Fortunately, Ruby provides an easy way to ship functionality that would prevent this. Developers may pass the block, instead of a method name, that would get called on the object. Therefore, in case of need for some special actions user could define configuration like this:

```
Administer::Config.set_display_name Category do
  name[0..10]
end
```

Seems to be much cleaner. Everything that belongs to the plugin is handled apart from the application. However, if more options would be handled it starts to gets a little messy:

```
Administer::Config.set_display_name Category do
  name.truncate(10)
end

Administer::Config.set_display_name(Post, :title)
Administer::Config.set_smth_else(Post, :smth_else
```

A lot of repeated code and it is easy to imagine what would happen if someone would like to define for example 6 options for one model. He would have to write 6 times almost the same thing with just changing name of option and method or block for that option.

So the search for the most suitable solution is still on. But, remembering that it is very dynamic environment, some smart outcome can be elaborated. After some deliberations, an idea of Domain Specific Language appeared. It could look like this:

```
Admnister::Config.configure do
  model Post do
    association_display :title
  end

  model Category do
    association_display do
      name.truncate[0..10]
    end
  end
end
```

### Implementation

The great thing is that it is not hard at all to implement such DSL in Ruby. In order to do this the only thing needed is understanding the power of blocks and usage of `eval` function.

The root class is `Administer::Config`. It provides a class method `configure` that takes one parameter - a block. The only thing that this function does is it creates an object of `Administer::Config` by calling a constructor and passing the block, and after the object is created it is assigned to the class variable so that the configuration for application is held in one place for entire object space.

The real work is done in constructor of `Administer::Config`. Although the code is not very long:

```
def initialize(&block)
  @model_configs = {}
  self.instance_eval &block
end
```

it shows the power of dynamic programming and to be specific it utilizes the `instance_eval` method. What this method does is it executes the block in context of the object it has been called on.

So, if there is such piece of code:

```
Admnister::Config.configure do
  model Post do
    association_display :title
  end
end
```

first it creates the object of `Administer::Config` and then from line 2 the code is executed in context of that newly created object. In other words, that code just calls the `model` in the config object and passes two parameters - class `Post` and a block.

Then, the situation repeats as the `model` method creates an instance of class called `Administer::ModelConfig` and the constructor again takes the block and executes it in context of that object. For further details I redirect to the code of application in order to dig deeper into its construction.

## 6.6 Routing, Controllers and Html Templates

Above has been described internals of the plugin. Now it is time for the more tangible part - the part that makes it working on a web and talks to the user via web interface. This part consists of three elements

- Routing
- Controllers
- Html Templates

### 6.6.1 Routing

The plugin adds following routes to the application (list shows path and HTTP method used for this route)

**GET /administer/dashboard** maps to `index` action of `DashboardController`. The resulting page shows the dashboard of administer, which at the time is just the list of available models with links to models' pages.

**GET /administer/entities** maps to `index` action of `EntitiesController`. Shows the list of all records of given model. The model name is passed in the parameter `model_name` so the route would be the following for model called `Category` - `/administer/entities?model_name=Category`. All routes that are mapped to `EntitiesController` need this paramaeter.

**GET /administer/entities/new** maps to `new` action of `EntitiesController`. Used for displaying form for new record of given model.

**POST /administer/entities** maps to `create` action of `EntitiesController`. Used for creating new records for given model. Parameters passed should be:

- `model_name` as in all entities rotues.
- hash named after the model (so for model `Category` the hash should be called `category`) that contains attributes for the newly created record.

**GET /administer/entities/:id** maps to `show` aciton of `EntitiesController`. It aims to show details of the record.

**GET /administer/entities/:id/edit** maps to `edit` action of `EntitiesController`. Used to display form for editing record. The `:id` variable should contain the id of record that is supposed to be edited. Example route for model `Category` and id 25: `/administer/entities/25/edit?model_name=Category`.

**PUT /administer/entities/:id** maps to `update` action of `EntitiesController`. Parameters are the same as for the route for `create` action. This route aims to update the record with given attributes.

**DELETE /administer/entities/:id** maps to `destroy` action of `EntitiesController`. It deletes queried record from the database.

The routes for entities follow guidelines for REST[4] architecture. This means that it uses rather vocabulary of HTTP for identifying actions instead of elaborating some

[4]Representational State Transfer

fancy routes to map them to appropriate actions. This is why, the difference between `show` and `update` or `destroy` action is just change of HTTP method from `GET` to `PUT` or `DELETE` instead of having to routes `/administer/entities/:id/show` and `/administer/entities/:id/update` or `/administer/entities/:id/delete`.

As it can be observed - RESTful architecture encourages to have single URI for managing particular resource. `Edit` and `new` actions have their own URIs as they are not in fact managing the resource - they are only needed for the purpose of displaying appropriate forms to users. It is hardly possible, that the user of application would craft appropriate queries and parameters himself. But if the user of the application would be another application it would not need that additional forms and could handle the resource it needs with single URI.

As a result, the administer part of the application could be easily turned into REST Web Service. Except for normal HTML response, it could be easily extended to send out JSON or XML responses.

### 6.6.2 Controllers

As it can be observed in the routing chapter, there are 2 controllers in administer plugin:

**DashboardController** handles the actions when the user is not managing any particular model. At the moment it is only `index` action that displays a list of all possible models.

**EntitiesController** handles user actions when the use is managing models.

Both of those controllers are subclasses of `Administer::ApplicationController`.

#### ApplicationController

The `Administer::ApplicationController` is the base controller for all of administer controllers. Thanks to this fact, it was possible to set up filter that would be active all over administer part of the application. This filter would add authentication functionality that can be configured in `Administer::Config`.

If a user defines a method for authentication, administer would call it before every request for administer part of the application. If any redirect would occur there, it would not proceed with actions of administer controllers. Therefore, the developer may define a method that checks whether appropriate user is logged in and if not it

would redirect to login page at the same time preventing administration section for unauthorized access.

### DashboardController

As said above, the functionality of `DashboardController` at the moment is just delivering the index action which, in fact, does not do anything in particular except for rendering the index view.

### EntitiesController

`EntitiesController` is the heart of administer web application. Its responsibility is to handle managing of models. As seen in routes, it provides following actions:

- index
- show
- new
- create
- edit
- update
- destroy

The names of the actions are self-explanatory. For all actions there is a filter introduced called `set_model`, which is used to set up a model before action is started. That way it was unnecessary to write the same code for every action. The filter takes the `model_name` parameter from HTTP query and build a `Administer::Model` based on that.

For actions that include forms, the controller prepares the list of fields (as described in Fields chapter) and puts it in instance variable `@fields`. It is worth to mention, that all instance variables of controller are accessible from view.

### 6.6.3 Views

#### Technology

The views have been developed using Haml[5] templating language. It has been chosen over standard ERB because of its cleanliness and conciseness. It requires much less code to produce the same output than ERB. For example[6] ERBs code:

```
<div id="profile">
  <div class="left column">
    <div id="date"><%= print_date %></div>
    <div id="address"><%= current_user.address %></div>
  </div>
  <div class="right column">
    <div id="email"><%= current_user.email %></div>
    <div id="bio"><%= current_user.bio %></div>
  </div>
</div>
```

can be transfered to Haml like this:

```
#profile
  .left.column
    #date= print_date
    #address= current_user.address
  .right.column
    #email= current_user.email
    #bio= current_user.bio
```

Much less code plus it automatically closes html tags, so it prevents the situation that somewhere in the partial a `div` has not been closed and that affects totally different `div` somewhere in the layout. Debugging of such errors is very tedious and troublesome.

#### Structure of views

The most outer part of the webpage is held in the layout file that is loaded for every action in every controller. So the layout consists of all tags that appear on every web page like `head` section, opening and closing of `body` tag and so on. The layout has the `yield` clause. This is the place where templates of particular actions are put in.

[5] http://haml-lang.com/
[6] Examples from HAML website

Every action has its own template named after the name of the action. If some part of the template maybe extracted and used in few places it is put in partial. Partial are named with underscore at the end (eg. `_partial.html.haml`). Then the partial can be included in the template by using `render "path_to_partial"`.

### Generating of forms

The most interesting parts of forms is generating of forms for models. As said before, the controller puts the list of fields for given model in `@fields` instance variable. The `edit` and `new` templates first create a form using `form_for` method. It just creates the form tag and passes the reference to form builder to the block. Then the routine for creating a form starts.

The list of fields is iterated and for every field `render field.partial` method is called. The `Administer::Fields::Base` class defines the partial method so that it returns the string - a path to the partial. The path is calculated by having the standard path `administer/fields/` and then adding the name of the current class trasnformed to lowercase with underscores to the end of it. `HasMany` becomes `"administer/fields/has_many"`. Then, partials for all available fields are defined in that path. Every partial defines appropriate html tags for given field.

### Styling

The administer plugin does not ship any fancy layout. The markup uses appropriate ids and classes so that it can be easily customized by the end user. This makes it stand out from the collection of available plugins as all of them impose some layout.

Nevertheless, even without setting up any CSS rules the administer plugin looks clean, readable and is fully usable. This is because it uses semantic markup, which means that it does not use arbitrary tags but those that are the most suitable for the part of the website they describes. Therefore, even if the web browser uses its default rendering rules, the website would be well arranged.

## 6.7 Discussion

### 6.7.1 General outcome

In general, the project has turned out to be a success. The plugin is functional and works as intended. Moreover, the most important aim of making it working out of the box (with the little configuration that has been discussed before) has been met.

After creating a Rails application and having `administer` in the Gemfile the application gets `/administer` path and the user is welcomed with screen seen in figure 6.5.

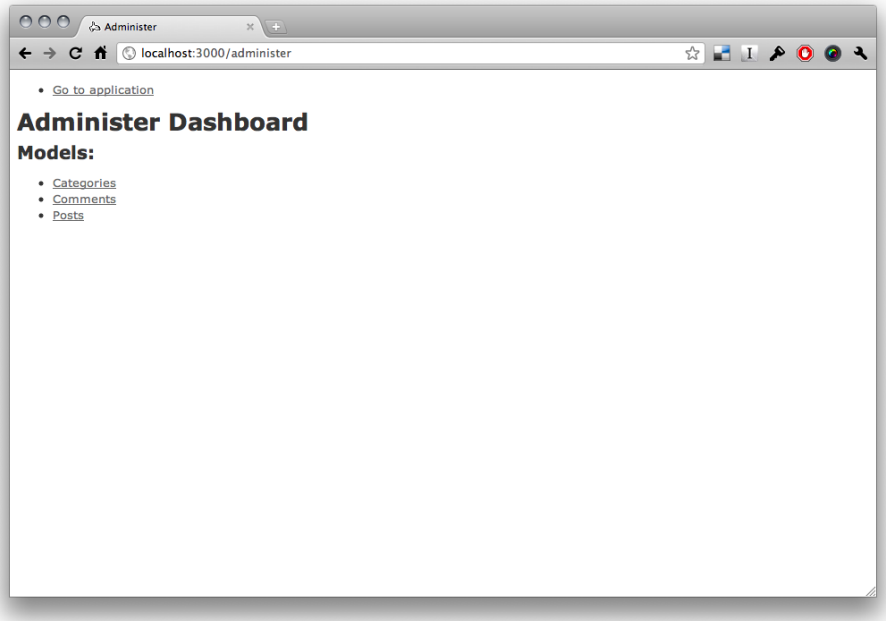



Figure 6.5: Administer Dashboard

Then while browsing the plugin, the user may browse (fig. 6.6 p. 74) and manage(fig. 6.7 p. 74) his models. All actions that have been established at the beginning of the project: browsing, creating, editing and deleting of model records are available and work as intended.

### 6.7.2 Unsolved problems

There are still some unsolved problems in the application. Fortunately, none of them cause the plugin to be non-functional and they are rather wishes for the improvement.

First of all, it still in some places is bound to ActiveRecord. Though, it was not an aim of the project to make it totally independent of ActiveRecord and the plugin has

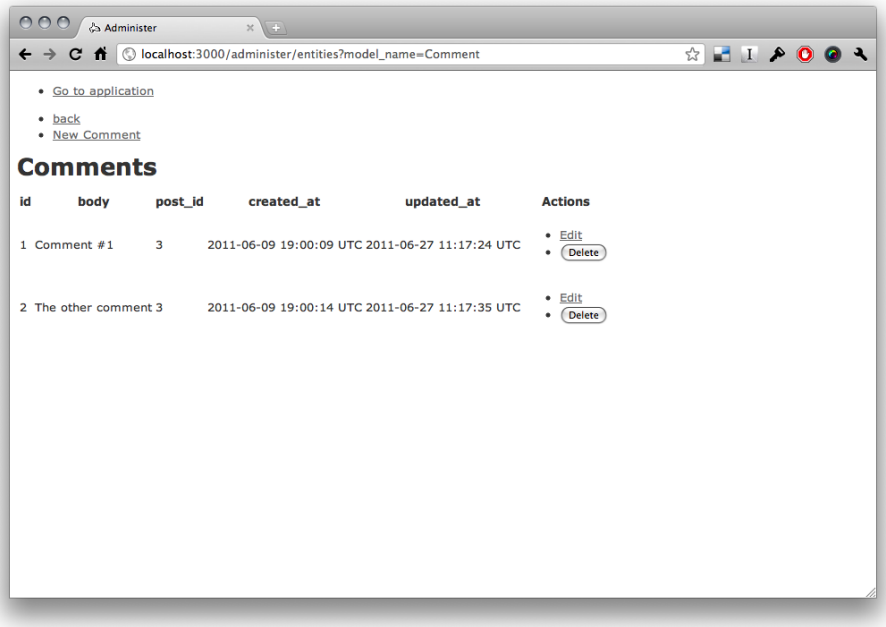



Figure 6.6: Administer list of records

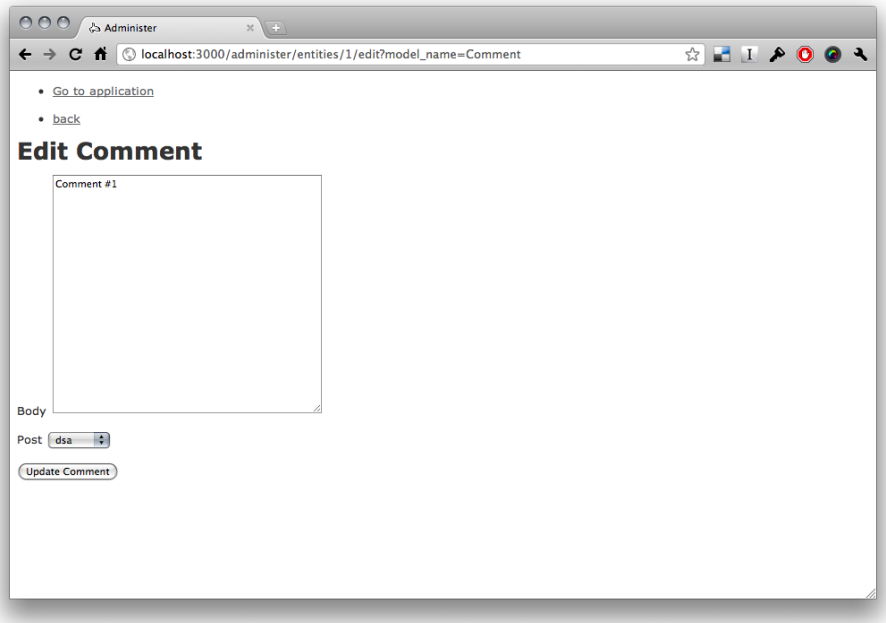



Figure 6.7: Administer: editing record

been developed for ActiveRecord based test application, it would be nice to decouple it and introduce totally modular binding to ORM.

Moreover, the configuration is very simple at the moment. It is really hard to guess all the options people would want. But, the implementation of configuration is pretty

open and it would be easy to introduce new options to the DSL once people start using it and posting requests.

The list of records could have been improved. As the rendering of forms have been thought through, and a lot effort has been put into making it work, the view that lists records is using very simple implementation. Probably some thought should be put there in order to make it more error prone and more easily customizable.

In the end, handling of `:through` associations would be a nice feature. Through associations are defined when one model is associated to another one via some intermediate model. Such associations are not yet handled here as some additional research would need to be conducted in order to find out how people would like to use it.

# Chapter 7

# Conclusions

As mentioned in chapter 6.7.1 about the outcome of the project (page 73) the project turned out to be successful.

One of my biggest concerns while I was starting the project was that the application needs to be universal and usable for all kind of purposes. Fortunately, it turned out that if enough of work and study is put into this, it is not only doable but also very enjoyable.

The plugin is far from being fully complete. Although it has all the functionalities that were set up in the beginning, it still can get a lot of improvements. Probably, the state when the developer says that this tool is total and does not need anything else in fact never happens. Needs of people would change and therefore new feature requests would emerge. Moreover, Rails framework is still evolving and the only fact of keeping the plugin up to date with the newest versions of the framework would take a lot of work.

Fortunately, the code base has been developed in a way that should be easily extendable. The fact that the development of the plugin has been driven by tests forced the design of code so that it is easy to change in one place without breaking it somewhere else. And if something breaks it would draw immediate attention as some of the test would fail.

There are already few features that probably would get added to the plugin in the nearest feature (as described in chapter 6.7.2 p. 73). I intend to make this plugin as comprehensive as possible.

As I would start to spreading the word about the plugin throughout the community, I hope for a lot constructive criticism and feature requests. Hopefully, this project would start to fulfill needs of Ruby on Rails developers and would make their life easier.

# Bibliography

[1] Paolo Perrotta, *Metaprogramming Ruby: Program Like the Ruby Pros*, Pragmatic Bookshelf, 2010.

[2] Gregory T. Brown, *Ruby Best Practices*, O'Reilly Media, Inc, 2009.

[3] "Ruby documentation", `http://www.ruby-doc.org/core/`, 2011.

[4] David Flanagan and Yukihiro Matsumoto, *The Ruby Programming Language*, O'Reilly Media, Inc, 2008.

[5] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North, *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*, Pragmatic Bookshelf, 2009.

[6] Steve Freeman and Nat Pryce, *Growing Object-Oriented Software, Guided By Tests*, Pearson Education, 2010.

[7] Mike Cohn, *Succeding with Agile: Software Development using Scrum*, Addison Wesley, 2009.

[8] Leonard Richardson and Sam Ruby, *RESTful Web Services*, O'Reilly Media, Inc, 2007.