Plug-in application for automatic generation of administration panel

Piotr Jakubowski

May 23, 2011

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Description

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.

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.

1.2 Existing Solutions

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.

1.2.1 Active Scaffold[2]

According to Ruby Toolbox[3] it is the most popular plugin for admin interface generation. Unfortunately, it seems like the days of glory for this gem are long

[1] Create Read Update Delete

[2] http://activescaffold.com/

[3] http://ruby-toolbox.com/categories/rails_admin_interfaces.html

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 1.1 and 1.2

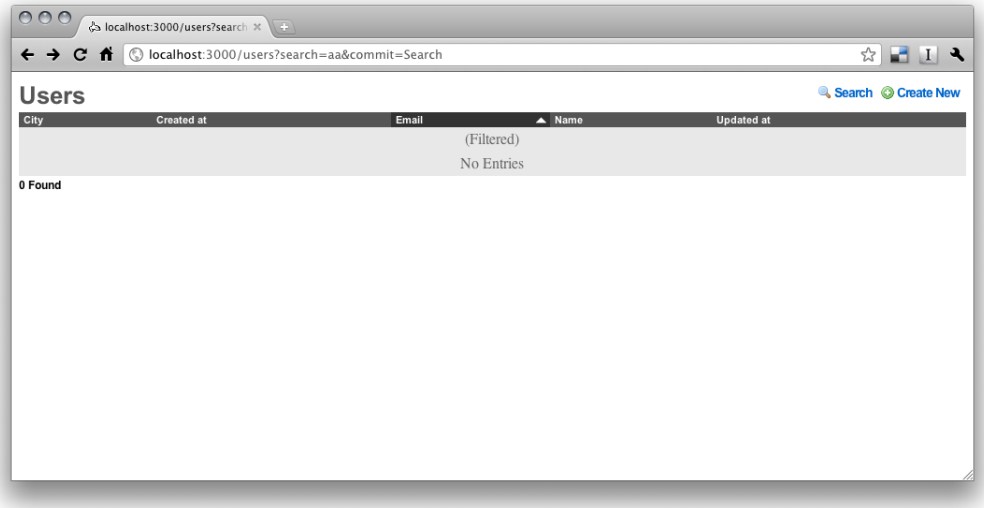

Figure 1.1: Active Scaffold

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.

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:

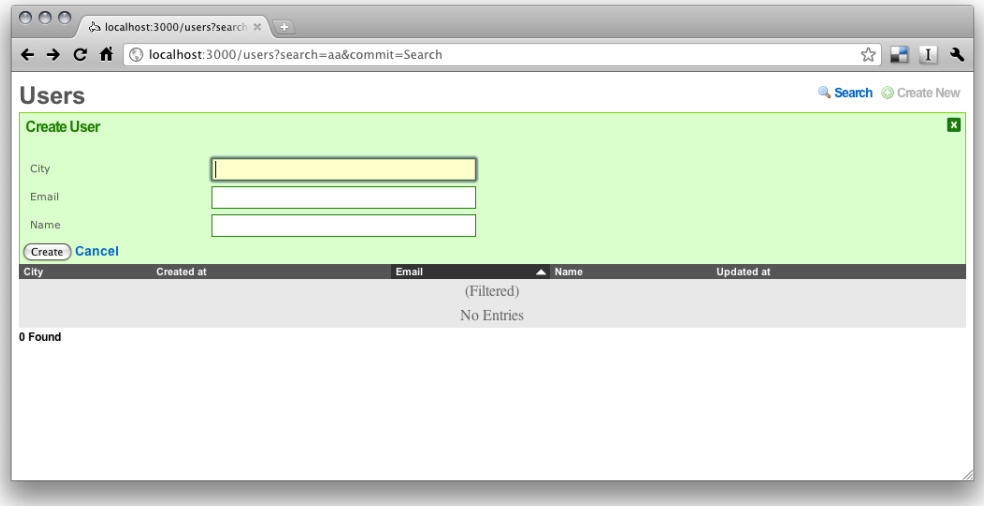

Figure 1.2: Automatically generated form in Active Scaffold

- 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.

1.2.2 Typus[4]

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 1.3 and 1.4.

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

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

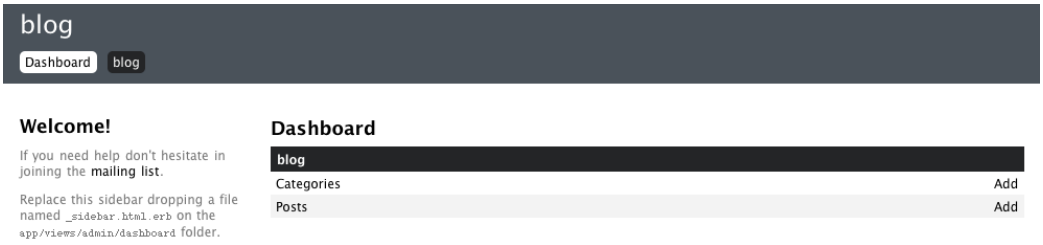

Figure 1.3: Typus main page

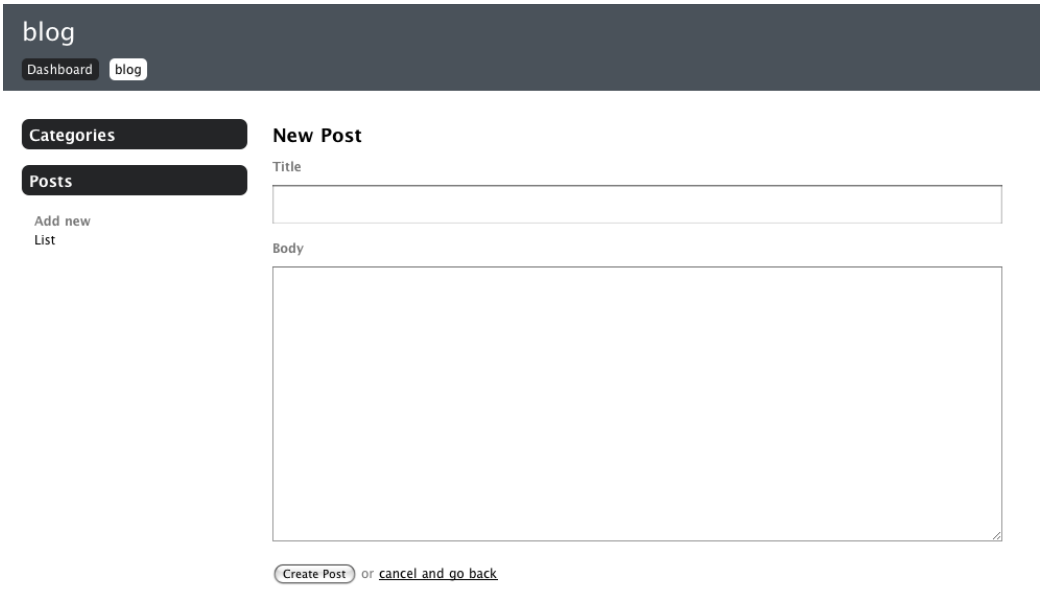

Figure 1.4: Automatically generated form in Typus

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.

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.

1.2.3 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 1.5 presents what more or less we would see after trying it out.

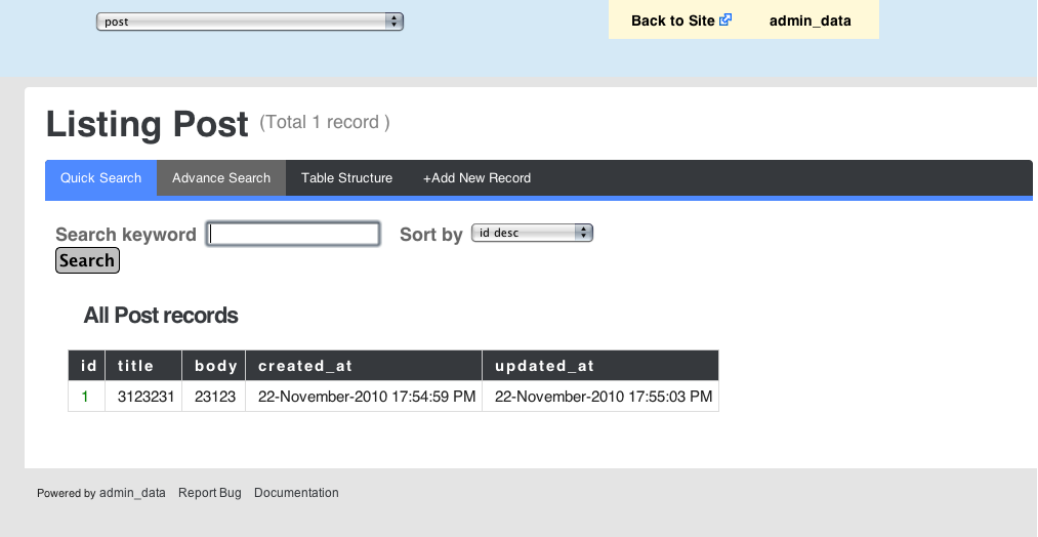

Figure 1.5: Admin data

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 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.

1.2.4 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[5]. 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[6] gem, which takes care of users authentication, as rails_admin's default option is to make people to log in to the administration panel.

What we get after installation is very nice and user-friendly panel(figure 1.6). 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.

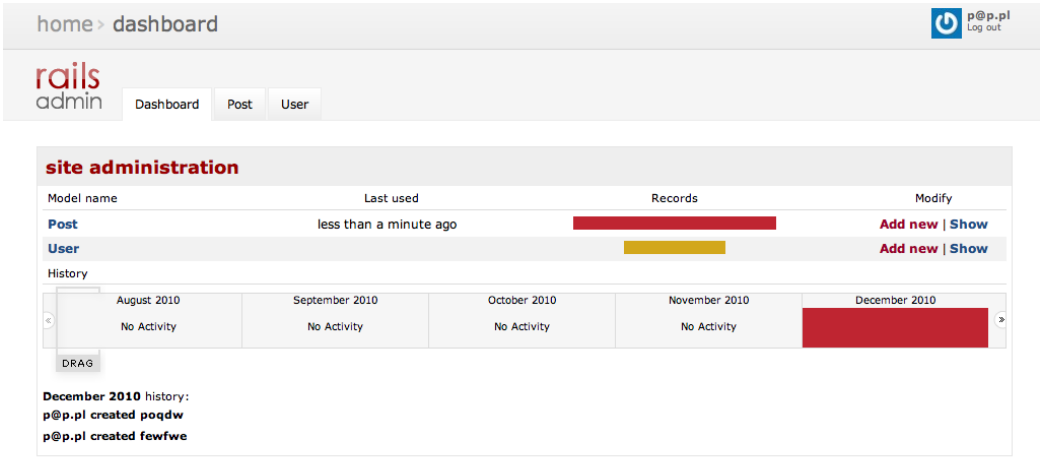

Figure 1.6: rails_admin dashboard

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.

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

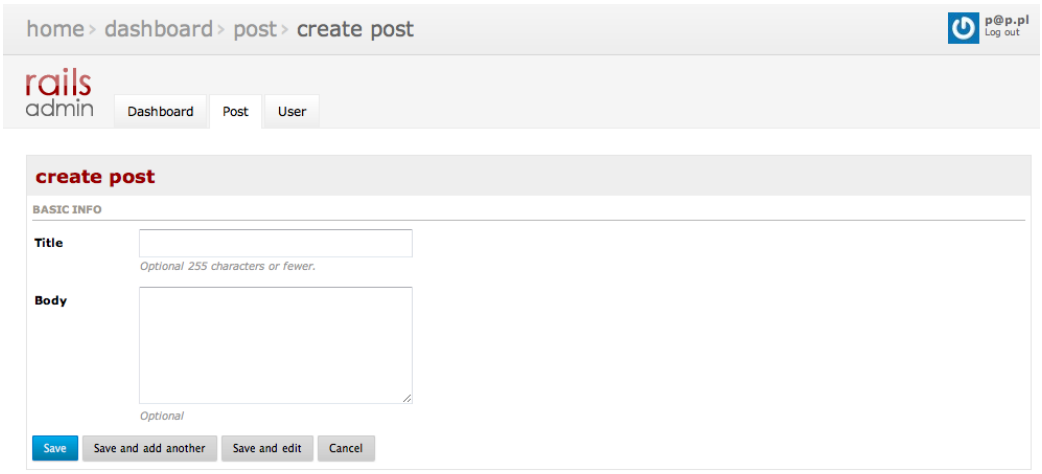

Figure 1.7: Form generated in rails_admin

Definitely, ideas used in that project are very interesting and may be very useful.

Chapter 2

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.

2.1 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.
- Configuration of the panel regarding which models or model fields should be visible.

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

2.2 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. Mock-ups of user interface of the plugin are depicted in figures 2.1, 2.2 and 2.3

Figure 2.1: Mockup of dashboard page of the plugin

2.3 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

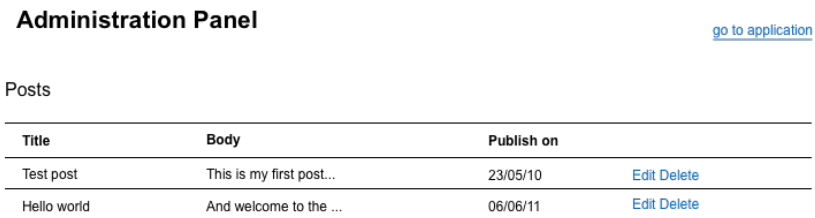

Figure 2.2: Mockup of page for listing records of given model

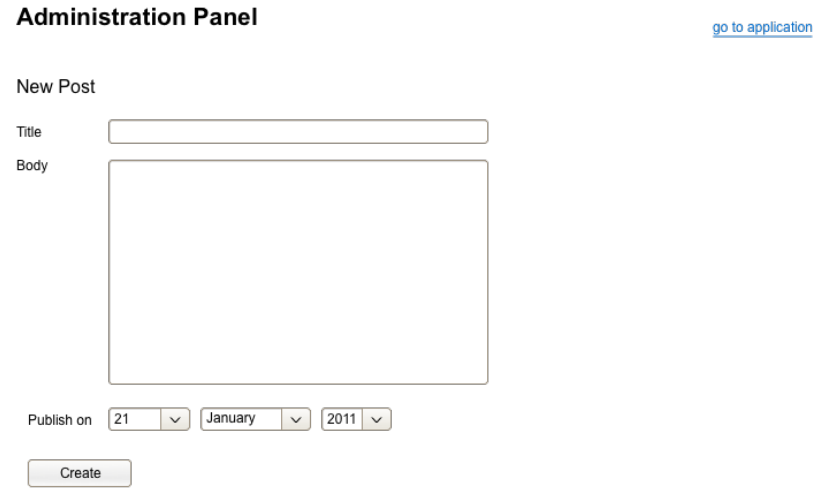

Figure 2.3: Example page for creation or edition of model record

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 3

Research

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.

3.1 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.

3.1.1 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

That explains the idea behind the project.

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]).

3.1.2 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

3.1.3 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[2].

3.2 RubyGems

3.2.1 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'
```

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.

3.2.2 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

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

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.

3.3 Ruby on Rails

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.4 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.

3.4.1 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.

3.4.2 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 informa-

tion 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 4

Solution development

4.1 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.

4.1.1 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/administ
```

4.1.2 Gemspec

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

```
$ 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/administ
```

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

Bibliography

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

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