**Ruby** **on** **Rails** **-** **Introduction**

**What is Ruby?\_**

Before we ride on Rails, let us recapitulate a few points of Ruby, which is the base of Rails.

Ruby is the successful combination of:

• Smalltalk's conceptual elegance,

• Python's ease of use and learning, and

• Perl's pragmatism.

Ruby is

• A high-level programming language.

• Interpreted like Perl, Python, Tcl/TK.

• Object-oriented like Smalltalk, Eiffel, Ada, Java.

**Why Ruby?\_**

Ruby originated in Japan and now it is gaining popularity in US and Europe as well. The following factors contribute towards its popularity:

• Easy to learn

• Open source (very liberal license)

• Rich libraries

• Very easy to extend

• Truly object-oriented

• Less coding with fewer bugs

• Helpful community

Although we have many reasons to use Ruby, there are a few drawbacks as well that you may have to consider before implementing Ruby:

• Performance Issues - Although it rivals Perl and Python, it is still an interpreted language and we cannot compare it with high-level programming languages like C or C+ +.

• Threading model - Ruby does not use native threads. Ruby threads are simulated in the VM rather than running as native OS threads.

**Ruby** **on** **Rails**

**Sample Ruby Code\_**

Here is a sample Ruby code to print "Hello Ruby"

# The Hello Class class Hello

def initialize( name )

@name = name.capitalize end

def salute

puts "Hello #{@name}!" end end

# Create a new object h =

Hello.new("Ruby")

# Output "Hello Ruby!" h.salute

**Embedded Ruby\_**

Ruby provides a program called ERb (Embedded Ruby), written by *Seki* *Masatoshi.* ERb allows you to put Ruby codes inside an HTML file. ERb reads along, word for word, and then at a certain point, when it encounters a Ruby code embedded in the document, it starts executing the Ruby code.

You need to know only two things to prepare an ERb document:

• If you want some Ruby code executed, enclose it between **<%** and **%>**.

• If you want the result of the code execution to be printed out, as a part of the output, enclose the code between **<%=** and **%>**.

Here's an example. Save the code in erbdemo.rb file. Note that a Ruby file will have an extension **.rb**:

<% page\_title = "Demonstration of ERb" %>

<% salutation = "Dear programmer," %>

<html>

<head>

<title><%= page\_title %></title>

</head>

<body>

**Ruby** **on** **Rails**

<p><%= salutation %></p>

<p>This is an example of how ERb fills out a template.</p>

</body>

</html>

Now, run the program using the command-line utility erb.

c:\ruby\>erb erbdemo.rb This will produce the following result:

<html>

<head>

<title>Demonstration of ERb</title>

</head>

<body>

<p>Dear programmer,</p>

<p>This is an example of how ERb fills out a template.</p>

</body>

</html>

**What is Rails?\_**

• An extremely productive web-application framework.

• Written in Ruby by David Heinemeier Hansson.

• You could develop a web application at least ten times faster with Rails than you could with a typical Java framework.

• An open source Ruby framework for developing database-backed web applications.

• Your code and database schema are the configuration!

• No compilation phase required.

**Full** **Stack** **Framework**

• Includes everything needed to create a database-driven web application, using the Model-View-Controller pattern.

• Being a full-stack framework means all the layers are built to work seamlessly together with less code.

• Requires fewer lines of code than other frameworks.

**Ruby** **on** **Rails**

**Convention** **overConfiguration**

• Rails shuns configuration files in favor of conventions, reflection, and dynamic runtime extensions.

• Your application code and your running database already contain everything that Rails needs to know!

**Rails Strengths\_**

Rails is packed with features that make you more productive, with many of the following features building on one other.

**Metaprogramming:** Other frameworks use extensive code generation from

scratch. Metaprogramming techniques use programs to write programs. Ruby is one of the best languages for metaprogramming, and Rails uses this capability well. Rails also uses code generation but relies much more on metaprogramming for the heavy lifting.

**Active** **Record:** Rails introduces the Active Record framework, which saves objects to the database. The Rails version of the Active Record discovers the columns in a database schema and automatically attaches them to your domain objects using metaprogramming.

**Convention** **over** **configuration:** Most web development frameworks for .NET or Java force you to write pages of configuration code. If you follow the suggested naming conventions, Rails doesn't need much configuration.

**Scaffolding:** You often create temporary code in the early stages of development to help get an application up quickly and see how major components work together. Rails automatically creates much of the scaffolding you'll need.

**Built-in** **testing:** Rails creates simple automated tests you can then extend. Rails also provides supporting code called harnesses and fixtures that make test cases easier to write and run. Ruby can then execute all your automated tests with the rake utility.

**Three** **environments:** Rails gives you three default environments: development, testing, and production. Each behaves slightly differently, making your entire software development cycle easier. For example, Rails creates a fresh copy of the Test database for each test run.

**2.** **Ruby** **on** **Rails** **–Installation**

To develop a web application using Ruby on Rails Framework, you need to install the following software -

• Ruby

• The Rails Framework

• A Web Server

• A Database System

We assume that you already have installed a Web Server and a Database System on your computer. You can use the WEBrick Web Server, which comes with Ruby. Most websites however use Apache or lightTPD web servers in production.

Rails works with many database systems, including MySQL, PostgreSQL, SQLite, Oracle, DB2 and SQL Server. Please refer to a corresponding Database System Setup manual to set up your database.

Let's look at the installation instructions for Rails on Windows and Linux.

**Rails Installation on Windows\_**

Follow the steps given below for installing Ruby on Rails.

**Step** **1:** **Check** **Ruby** **Version**

First, check if you already have Ruby installed. Open the command prompt and type **ruby** **-v**. If Ruby responds, and if it shows a version number at or above 2.2.2, then type **gem** **--version**. If you don't get an error, skip **Install** **Ruby** step. Otherwise, we'll install a fresh Ruby.

**Step** **2:** **Install** **Ruby**

If Ruby is not installed, then download an installation package from [**rubyinstaller.org**](http://rubyinstaller.org/) Follow the **download** link, and run the resulting installer. This is an exe file **rubyinstaller-2.2.2.x.exe** and will be installed in a single click. It's a very small package, and you'll get RubyGems as well along with this package. Please check the **Release** **Notes** for more detail.

**Ruby** **on** **Rails**

**Help** **Contribute**

**Rubylnstaller** **About** **Download**

**The** **easy** **way** **to** **install** **Ruby** **on** **Windows**

**This** **i%** **A** **self-contained** **VV.ndows-bosed** **installer** **rti\*t** **incAjtfc^** **PHI**

**Ruby** language nn exocufcoo unrr'ri-nrTwinit impoitfl' •' **documentation** and AM*SV* **Download** **Add-ons**

Latest News Extras

**RutoylnatAlkf** **2** **0** **0-p645.** **2** **1** **6** **and** **2** **2** **2** **released** **<\*\*\*** **Pr/vw-««**

**Tne\*r#\*»** **retakes** **of** **JfeA/v** **»MimiUO>MVVyr** **|C\«** **-?01£-1&S6p** *2* **It** **Bf«a** *2* *2* **^**

**««** **Pue~~—~~a~~>~~CWH** **RobylntUlkf** **2** **2** **1** **released**

**Tne** **RuO>intlsar«** **daaoueoe** *of***Rubr** **2** **2** **t** **w»** **pa!** **lefcased** **cneawut** **tne** **0OMIC>\*7I** **sector** **I** **er** **live** **\*»** **an** **mew** **uunaaiei** **The**

**ter** **tfe** **ranters** **and** **Unary** **wtLjjft** **tMa**

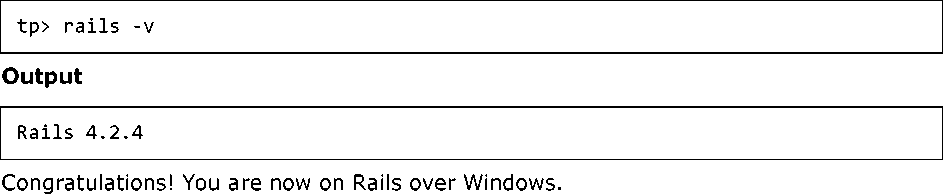
<•»•»» \*’■ O ■WWrtJI

**Step** **3:** **Install** **Rails**

**install** **Rails:** With Rubygems loaded, you can install all of Rails and its dependencies using the following command through the command line -

C:\> gem install rails

**Note:** The above command may take some time to install all dependencies. Make sure you are connected to the internet while installing gems dependencies.

**Ruby** **on** **Rails**

**Step** **4:** **Check** **Rails** **Version**

Use the following command to check the rails version.

**Rails Installation on Linux\_**

We are installing Ruby On Rails on Linux using **rbenv**. It is a lightweight Ruby Version Management Tool. The **rbenv** provides an easy installation procedure to manage various versions of Ruby, and a solid environment for developing Ruby on Rails applications.

Follow the steps given below to install Ruby on Rails using rbenv tool.

**Step** **1:** **Install** **Prerequisite** **Dependencies**

First of all, we have to install **git** **-** **core** and some ruby dependences that help to install Ruby on Rails. Use the following command for installing Rails dependencies using **yum**.

tp> sudo yum install -y git-core zlib zlib-devel gcc-c++ patch readline readline-devel libyaml-devel libffi-devel openssl-devel make bzip2 autoconf automake libtool bison curl sqlite-devel

**Step** **2:** **Install** **rbenv**

Now we will install rbenv and set the appropriate environment variables. Use the following set of commands to get rbenv for git repository.

tp> git clone git://github.com/sstephenson/rbenv.git .rbenv tp> echo

'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bash\_profile tp> echo 'eval

"$(rbenv init -)"' >> ~/.bash\_profile tp> exec $SHELL

tp> git clone git://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build

tp> echo 'export PATH=''$HOME/.rbenv/plugins/ruby-build/bin:$PATH'" >> ~/.bash\_profile

tp> exec $SHELL

**Ruby** **on** **Rails**

**Step** **3:** **Install** **Ruby**

Before installing Ruby, determine which version of Ruby you want to install. We will install Ruby 2.2.3. Use the following command for installing Ruby.

tp> rbenv install -v 2.2.3

Use the following command for setting up the current Ruby version as default.

tp> rbenv global 2.2.3

Use the following command to verify the Ruby version.

tp> ruby -v

**Output**

ruby 2.2.3p173 (2015-08-18 revivion 51636) [X86\_64-linux]

Ruby provides a keyword **gem** for installing the supported dependencies; we call them**gems**. If you don't want to install the documentation for Ruby-gems, then use the following command.

tp> echo "gem: --no-document" > ~/.gemrc

Thereafter, it is better to install the Bundler gem, because it helps to manage your application dependencies. Use the following command to install bundler gem.

tp> gem install bundler

**Step** **4:** **Install** **Rails**

Use the following command for installing Rails version 4.2.4.

tp> install rails -v 4.2.4

Use the following command to make Rails executable available.

tp> rbenv rehash

Use the following command for checking the rails version.

tp> rails -v

**Output**

tp> Rails 4.2.4

Ruby on Rails framework requires JavaScript Runtime Environment (Node.js) to manage the features of Rails. Next, we will see how we can use Node.js to manage Asset Pipeline which is a Rails feature.

**Ruby** **on** **Rails**

**Step** **5:** **Install** **JavaScript** **Runtime**

Let us install install Node.js from the Yum repository. We will take Node.js from EPEL yum repository. Use the following command to add the EPEL package to the yum repository.

tp> sudo yum -y install epel-release

Use the following command for installing the Node.js package.

tp> sudo yum install nodejs

Congratulations! You are now on Rails over Linux.

**Step** **6:** **Install** **Database**

By default, Rails uses sqlite3, but you may want to install MySQL, PostgreSQL, or other RDBMS. This is optional; if you have the database installed, then you may skip this step and it is not mandatory that you have a database installed to start the rails server. For this tutorial, we are using PostgreSQL database. Therefore use the following commands to install PostgreSQL.

tp> sudo yum install postgresql-server postgresql-contrib

Accept the prompt, by responding with a **y**. Use the following command to create a PostgreSQl database cluster.

tp> sudo postgresql-setup initdb

Use the following command to start and enable PostgreSQL.

tp> sudo systemctl start postgresql tp> sudo systemctl

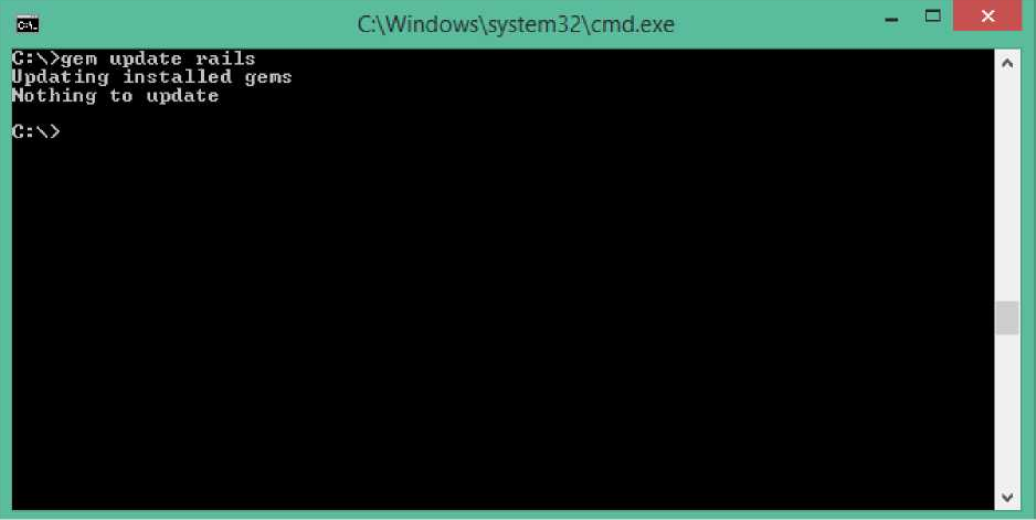
enable postgresql

**Keeping Rails Up-to-Date\_**

Assuming you have installed Rails using RubyGems, keeping it up-to-date is relatively easy. We can use the same command in both Windows and Linux platform. Use the following command -

tp> gem update rails **Output**

The following screenshot shows a Windows command prompt. The Linux terminal also provides the same output.

**Ruby** **on** **Rails**

This will automatically update your Rails installation. The next time you restart your application, it will pick up this latest version of Rails. While using this command, make sure you are connected to the internet.

**Installation Verification\_**

You can verify if everything is set up according to your requirements or not. Use the following command to create a demo project.

tp> rails new demo

**Output**

It will generate a demo rail project; we will discuss about it later. Currently we have to check if the environment is set up or not. Next, use the following command to run WEBrick web server on your machine.

**Ruby** **on** **Rails**

tp> cd demo tp> rails server

It will generate auto-code to start the server

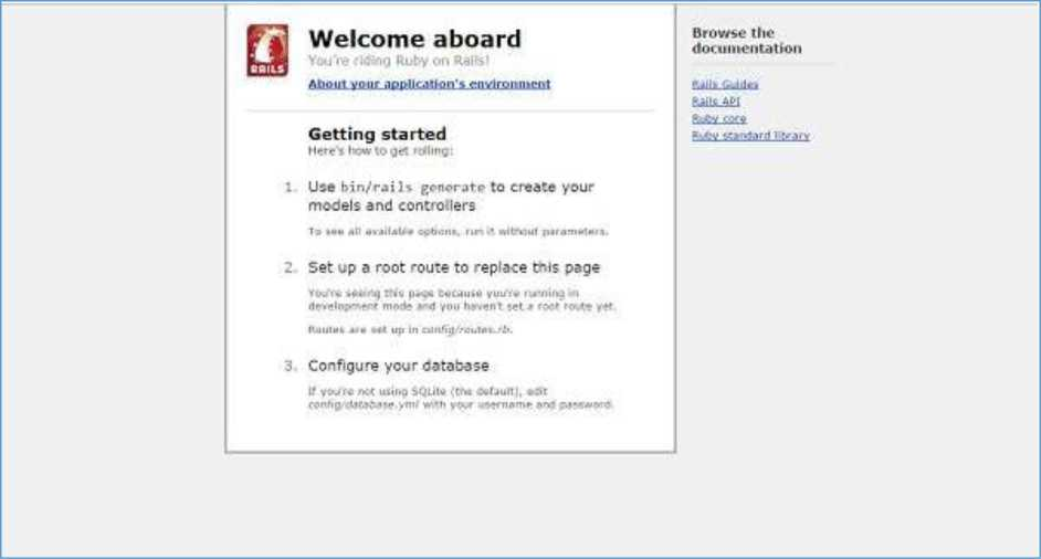
S rails server

DL is deprecated, please use Fiddle => Booting WEBrick

*=>* Rails 4,2,3 application starting in development on <http://locaihost:3000>*=>* Run 'rails server -h' for more startup options => Ctrl-C to shutdown server

Now open your browser and type the following -

<http://localhost:3000>

It should display a message, something like, "Welcome aboard" or "Congratulations".

**3.** **Ruby** **on** **Rails** **-Framework**

A framework is a program, set of programs, and/or code library that writes most of your application for you. When you use a framework, your job is to write the parts of the application that make it do the specific things you want.

When you set out to write a Rails application, leaving aside the configuration and other housekeeping chores, you have to perform three primary tasks:

**•** **Describe** **and** **model** **your** **application's** **domain:** The domain is the universe of your application. The domain may be a music store, a university, a dating service, an address book, or a hardware inventory. So here you have to figure out what's in it, what entities exist in this universe and how the items in it relate to each other. This is equivalent to modeling a database structure to keep the entities and their relationship.

**•** **Specify** **what** **can** **happen** **in** **this** **domain:** The domain model is static; you have to make it dynamic. Addresses can be added to an address book. Musical scores can be purchased from music stores. Users can log in to a dating service. Students can register for classes at a university. You need to identify all the possible scenarios or actions that the elements of your domain can participate in.

**•** **Choose** **and** **design** **the** **publicly** **available** **views** **of** **the** **domain:** At this point, you can start thinking in Web-browser terms. Once you've decided that your domain has students, and that they can register for classes, you can envision a welcome page, a registration page, and a confirmation page, etc. Each of these pages or views shows the user how things stand at a certain point.

Based on the above three tasks, Ruby on Rails deals with a Model/View/Controller (MVC) framework.

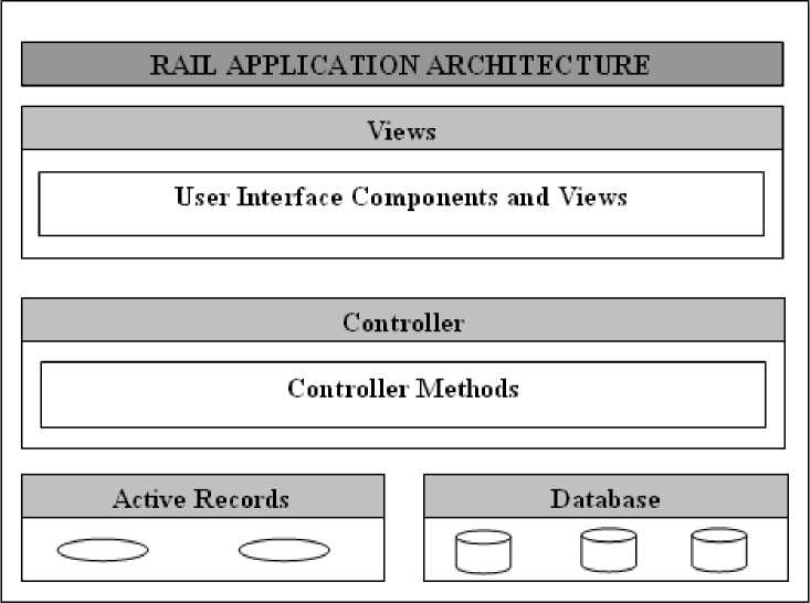
**Ruby on Rails MVC Framework\_**

The **M**odel **V**iew **C**ontroller principle divides the work of an application into three separate but closely cooperative subsystems.

**Model** **(ActiveRecord)**

It maintains the relationship between the objects and the database and handles validation, association, transactions, and more.

This subsystem is implemented in ActiveRecord library, which provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records. Ruby method names are automatically generated from the field names of database tables.

**Ruby** **on** **Rails**

**View** **(ActionView)**

It is a presentation of data in a particular format, triggered by a controller's decision to present the data. They are script-based template systems like JSP, ASP, PHP, and very easy to integrate with AJAX technology.

This subsystem is implemented in ActionView library, which is an Embedded Ruby (ERb) based system for defining presentation templates for data presentation. Every Web connection to a Rails application results in the displaying of a view.

**Controller** **(ActionController)**

The facility within the application that directs traffic, on the one hand, querying the models for specific data, and on the other hand, organizing that data (searching, sorting, massaging it) into a form that fits the needs of a given view.

This subsystem is implemented in ActionController, which is a data broker sitting between ActiveRecord (the database interface) and ActionView (the presentation engine).

**Pictorial Representation of MVC Framework\_**

Given below is a pictorial representation of Ruby on Rails Framework:

**Directory Representation of MVC Framework\_**

Assuming a standard, default installation over Linux, you can find them like this:

tp> cd /usr/local/lib/ruby/gems/1.8/gems

tp> ls

**Ruby** **on** **Rails**

You will see subdirectories including (but not limited to) the following:

• actionpack-x.y.z

• activerecord-x.y.z

• rails-x.y.z

Over a windows installation, you can find them like this:

tp>cd ruby\lib\ruby\gems\1.8\gems ruby\lib\ruby\gems\1.8\gems\>dir

You will see subdirectories including (but not limited to) the following:

• actionpack-x.y.z

• activerecord-x.y.z

• rails-x.y.z

ActionView and ActionController are bundled together under ActionPack.

ActiveRecord provides a range of programming techniques and shortcuts for manipulating data from an SQL database. ActionController and ActionView provides facilities for manipulating and displaying that data. Rails ties it all together.

**5.** **Ruby** **on** **Rails** **-** **Examples**

Subsequent chapters are based on the example given in this chapter. In this chapter, we will create a simple but operational online library system for holding and managing the books.

This application has a basic architecture and will be built using two ActiveRecord models to describe the types of data that is stored:

• Books, which describes an actual listing.

• Subject, which is used to group books together.

**Workflow for Creating Rails Applications\_**

A recommended workflow for creating Rails Application is as follows:

1. Use the rails command to create the basic skeleton of the application.

2. Create a database on the MySQL server to hold your data.

3. Configure the application to know where your database is located and the login credentials for it.

4. Create Rails Active Records (Models), because they are the business objects you'll be working with in your controllers.

5. Generate Migrations that simplify the creating and maintaining of database tables and columns.

6. Write Controller Code to put a life in your application.

7. Create Views to present your data through User Interface.

So, let us start with creating our library application.

**Creating an Empty Rails Web Application\_**

Rails is both a runtime web application framework and a set of helper scripts that automate many of the things you do when developing a web application. In this step, we will use one such helper script to create the entire directory structure and the initial set of files to start our Library System application.

• Go into ruby installation directory to create your application.

• Run the following command to create a skeleton for library application.

tp> rails library

This will create a subdirectory for the library application containing a complete directory tree of folders and files for an empty Rails application. Check a complete directory structure of the application. Check **Rails** **Directory** **Structure** for more detail.

**Ruby** **on** **Rails**

Most of our development work will be creating and editing files in the **library/app** subdirectories. Here's a quick rundown of how to use them:

• The *controllers* subdirectory is where Rails looks to find controller classes. A controller handles a web request from the user.

• The *views* subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.

• The *models* subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it dead simple.

• The *helpers* subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the model, view, and controller code small, focused, and uncluttered.

**Starting Web Server\_**

Rails web application can run under virtually any web server, but the most convenient way to develop a Rails web application is to use the built-in WEBrick web server. Let's start this web server and then browse to our empty library application:

This server will be started from the application directory as follows. It runs on port number 3000.

tp> cd ruby\library ruby\library\> ruby script/server This will start your WEBrick web server.

Now open your browser and browse to <http://127.0.0.1:3000>. If everything is gone fine, then you should see a greeting message from WEBrick, otherwise there is something wrong with your setting.

**What** **is** **Next?**

The next chapter explains how to create databases for your application and what is the configuration required to access these created databases.

Further, we will see what Rails Migration is and how it is used to maintain database tables.

**Ruby** **on** **Rails**

**13.** **Ruby** **on** **Rails** **-** **Scaffolding**

While you're developing Rails applications, especially those which are mainly providing you with a simple interface to data in a database, it can often be useful to use the scaffold method.

Scaffolding provides more than cheap demo thrills. Here are some benefits:

• You can quickly get code in front of your users for feedback.

• You are motivated by faster success.

• You can learn how Rails works by looking at the generated code.

• You can use scaffolding as a foundation to jumpstart your development.

**Scaffolding** **Example**

To understand *scaffolding,* let's create a database called **cookbook** and a table called **recipes**.

**Creating an Empty Rails Web Application\_**

Open a command window and navigate to where you want to create this **cookbook** web application. We have used c:\ruby. So, run the following command to create a complete directory structure.

C:\ruby> rails cookbook

**Setting Up the Database\_** Here is the way to create a database:

mysql> create database cookbook;

Query OK, 1 row affected (0.01 sec)

mysql> grant all privileges on cookbook.\* to

'root'@'localhost' identified by 'password';

Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;

Query OK, 0 rows affected (0.00 sec)

**Ruby** **on** **Rails**

To instruct Rails how to find the database, edit the configuration file c:\ruby\cookbook\config\database.yml and change the database name to cookbook. Leave the password empty. When you finish, it should look as follows:

development: adapter: mysql database: cookbook username: root password: [password]

host: localhost test:

adapter: mysql database: cookbook username: root password: [password] host:

localhost production: adapter: mysql database: cookbook username: root password:

[password] host: localhost

Rails lets you run in the development mode, test mode, or production mode, using different databases. This application uses the same database for each.

**The Generated Scaffold Code\_**

With the scaffold action, Rails generates all the code it needs dynamically. By running *scaffold* as a script, we can get all the code written to disk, where we can investigate it and then start tailoring it to our requirements.

So now, let's start once again to generate Scaffold code manually by using the scaffold helper script:

cookbook> rails generate scaffold recipe

**Ruby** **on** **Rails**

It generates auto-files as shown below

invoke active\_record

create db/inigrate/2fll50707112642\_create\_recipes.rb create app/models/recipe.rb

invoke test\_unit

create test/models/recipe test.rb create test/fixtures/recipes.yml

invoke resource\_route route resources :recipes invoke scaffold\_controller

create app/controllers/recipes\_controller.rb invoke erb

create app/views/recipes

create app/views/recipes/index,html.erb create app/views/recipes/edit. htinl . erb create app/views/recipes/show.html.erb create app/views/recipes/new.html.erb create app/views/recipes/ form,html.erb invoke test\_unit

create test/controllers/recipes controller test.rb invoke helper

create app/helpers/recipes helper.rb invoke test\_unit

invoke jbuilder

create app/views/recipes/index.ison.ibuilder create app/views/recipes/show.json.jbuilder invoke assets invoke coffee

create app/assets/i avascrjpts/recipes.coffee invoke scss

create app/assets/stvlesheets/recipes.scss invoke scss

create app/assets/stvlesheets/scaffolds.scss

**The Controller\_**

Let's look at the code behind the controller. This code is generated by the **scaffold** generator. If you open app/controllers/recipes\_controller.rb, then you will find something as follows:

class RecipesController < ApplicationController

before\_action :set\_recipe, only: [:show, :edit, :update, :destroy]

# GET /recipes

# GET /recipes.json def index

@recipes = Recipe.all end

**Ruby** **on** **Rails**

# GET /recipes/1

# GET /recipes/l.json def show

end

# GET /recipes/new def new

@recipe = Recipe.new end

# GET /recipes/1/edit def edit end

# POST /recipes

# POST /recipes.json def create

@recipe = Recipe.new(recipe\_params)

respond\_to do |format| if @recipe.save

format.html { redirect\_to @recipe, notice: 'Recipe was successfully created.' }

format.json { render :show, status: :created, location: @recipe } else

format.html { render :new }

format.json { render json: @recipe.errors, status: :unprocessable\_entity }

end

end

end

# PATCH/PUT /recipes/1

# PATCH/PUT /recipes/1.json def update

respond\_to do |format|

if @recipe.update(recipe\_params)

**Ruby** **on** **Rails**

format.html {

updated.' } redirect\_to @recipe, notice: 'Recipe was successfully

format.json {

else render :show, status: :ok, location: @recipe } format.html { render :edit }

format.json { render json: @recipe.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /recipes/1

# DELETE /recipes/1.json def destroy

@recipe.destroy respond\_to do |format|

format.html { redirect\_to recipes\_url, notice: 'Recipe was successfully destroyed.' }

format.json { head :no\_content } end end

private

# Use callbacks to share common setup or constraints between actions.

def set\_recipe

@recipe = Recipe.find(params[:id]) end

# Never trust parameters from the scary internet, only allow the white list through.

def recipe\_params

params.require(:recipe).permit(:tittle, :instructions) end

end

**Ruby** **on** **Rails**

When the user of a Rails application selects an action, e.g. "Show" - the controller will execute any code in the appropriate section - "def show" - and then by default will render a template of the same name - "show.html.erb". This default behavior can be overwritten.

The controller uses ActiveRecord methods such as find, find\_all, new, save, update\_attributes, and destroy to move data to and from the database tables. Note that you do not have to write any SQL statements, rails will take care of it automatically.

This single line of code will bring the database table to life. It will provide with a simple interface to your data, and ways of:

• Creating new entries

• Editing current entries

• Viewing current entries

• Destroying current entries

When creating or editing an entry, scaffold will do all the hard work like form generation and handling for you, and will even provide clever form generation, supporting the following types of inputs:

• Simple text strings

• Text areas (or large blocks of text)

• Date selectors

• Date-time selectors

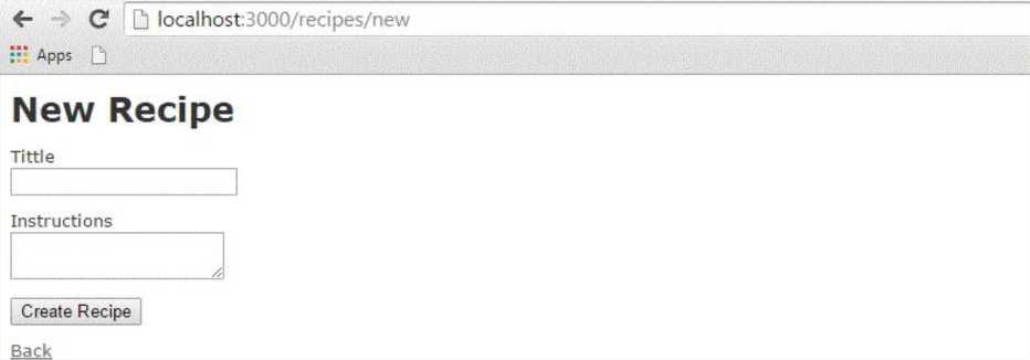
You can use Rails Migrations to create and maintain tables.

rake db:migrate RAILS\_ENV=development

Now, go to the cookbook directory and run the Web Server using the following command:

cookbook> rails server

Now, open a browser and navigate to <http://127.0.0.1:3000/recipe/new>. This will provide you a screen to create new entries in the recipes table. A screenshot is shown below:

**Ruby** **on** **Rails**

Once you press the **Create** button to create a new recipe, your record is added into the recipes table and it shows the following result:

**4-** **C** D localhost:3000/recipes/1 **•"** **Apps** **Q\_\_**

Recipe was successfully created.

**Tittle:** chicken **Instructions:** nice one Edit | Back

You can see the option to edit, show, and destroy the records. So, play around with these options.

You can also list down all the recipes available in the recipes table using the URL <http://127.0.0.1:3000/recipe/list>.

**Enhancing the Model\_**

Rails gives you a lot of error handling for free. To understand this, add some validation rules to the empty recipe model:

Modify app/models/recipe.rb as follows and then test your application:

class Recipe < ActiveRecord::Base

validates\_length\_of :title, :within ^1..20 validates\_uniqueness\_of :title,

:message ^"already exists" end

**Ruby** **on** **Rails**

These entries will give automatic checking.

**•** **validates\_length\_of** - the field is not blank and not too long.

**•** **validates\_uniqueness\_of** - duplicate values are trapped. Instead of the default Rails error message, we have given a custom message here.

**Alternative Way to Create Scaffolding\_**

Create an application as shown above and **The** **Generated** **Scaffold** **Code** as shown below

rails g scaffold Recipe tittle:string instructions:text

Above code generates the auto files with data base by using with sqlite3 with tittle and instruction column as shown below an image.

**invoke** **active\_record**

**create** **db/migrate/20158787112642 create recipes.rb** **create** **app/models/recioe.rb**

**invoke** **test\_unit**

**create** **test/models/recioe test.rb**

**create** **test/fixtures/recipes,yml resource\_route** **invoke** **resources** **:recipes** **scaffold\_controller**

**route** **app/controllers/recipes controller.rb erb** **invoke** **app/views/recipes**

**create** **app/yiews/recipes7index,htn'l.erb** **invoke** **app/views/recipes/edit.html.erb** **create** **app/views/recioes/show.html.erb** **create** **app/views/recioes/new,html,erb**

**create** **app/views/recipes/ form.html.erb test\_unit** **create** **test/controllers/recipes controller test.rb** **create** **helper**

**create** **app/helpers/recioes helper.rb test\_unit** **jbuilder** **invoke** **app/views/recipes/index.json.jbuilder**

**create** **aop/views/recioes/show. -json. jbuilder assets** **invoke** **coffee**

**create** **app/assets/iavascrjpts/recipes.coffee scss** **invoke** **app/assets/stylesheets/recipes.scss**

**invoke** **scss**

**create** **app/assets/stylesheets/scaffolds.scss** **create**

**invoke** **invoke** **create** **invoke** **create** **invoke** **create**

We need to migrate the data base by using below syntax

$ rake db:migrate RAILS\_ENV=development

**Ruby** **on** **Rails**

Finally run the application by using the following command line: rails server It will generate the result as shown above output images.

**The** **Views**

All the views and corresponding all the controller methods are created by **scaffold** command and they are available in the app/views/recipes directory.

**How** **Scaffolding** **is** **Different?**

If you have gone through the previous chapters, then you must have seen that we had created methods to list, show, delete and create data etc., but scaffolding does that job automatically.

**14.** **Ruby** **on** **Rails** **-** **Ajax** **on** **Rails**

Ajax stands for **A**synchronous **Ja**vaScript and **X**ML. Ajax is not a single technology; it is a suite of several technologies. Ajax incorporates the following:

• XHTML for the markup of web pages

• CSS for the styling

• Dynamic display and interaction using the DOM

• Data manipulation and interchange using XML

• Data retrieval using XMLHttpRequest

• JavaScript as the glue that meshes all this together

Ajax enables you to retrieve data for a web page without having to refresh the contents of the entire page. In the basic web architecture, the user clicks a link or submits a form. The form is submitted to the server, which then sends back a response. The response is then displayed for the user on a new page.

When you interact with an Ajax-powered web page, it loads an Ajax engine in the background. The engine is written in JavaScript and its responsibility is to both communicate with the web server and display the results to the user. When you submit data using an Ajax-powered form, the server returns an HTML fragment that contains the server's response and displays only the data that is new or changed as opposed to refreshing the entire page.

For a complete detail on AJAX you can go through our AJAX Tutorial

**How Rails Implements Ajax\_**

Rails has a simple, consistent model for how it implements Ajax operations. Once the browser has rendered and displayed the initial web page, different user actions cause it to display a new web page (like any traditional web application) or trigger an Ajax operation:

• **Some** **trigger** **fires:** This trigger could be the user clicking on a button or link, the user making changes to the data on a form or in a field, or just a periodic trigger (based on a timer).

• **The** **web** **client** **calls** **the** **server:** A JavaScript method, *XMLHttpRequest,* sends data associated with the trigger to an action handler on the server. The data might be the ID of a checkbox, the text in an entry field, or a whole form.

• **The** **server** **does** **processing:** The server-side action handler (Rails controller action) does something with the data and returns an HTML fragment to the web client.

• **The** **client** **receives** **the** **response:** The client-side JavaScript, which Rails creates automatically, receives the HTML fragment and uses it to update a specified part of the current page's HTML, often the content of a <div> tag.

**Ruby** **on** **Rails**

These steps are the simplest way to use Ajax in a Rails application, but with a little extra work, you can have the server return any kind of data in response to an Ajax request, and you can create custom JavaScript in the browser to perform more involved interactions.

**AJAX Example\_**

While discussing rest of the Rails concepts, we have taken an example of Library. There we have a table called **subject** and we have added few subjects at the time of Migration. Till now, we have not provided any procedure to add and delete subjects in this table.

In this example, we will provide, list, show, and create operations on subject table. If you don't have any understanding on Library Info System explained in the previous chapters, then we would suggest you to go through the previous chapters first and then continue with AJAX on Rails.

**Creating** **an** **Application**

Let us start with the creation of an application. It will be done as follows:

rails new ponies

The above command creates an application, now we need to call the app directory using with cd command. It will enter in to an application directory then we need to call a scaffold command. It will be done as follows

rails generate scaffold Pony name:string profession:string

The above command generates the scaffold with name and profession column. We need to migrate the data base as follows command

rake db:migrate

Now Run the Rails application as follows command

rails s

Now open the web browser and call a url as <http://localhost:3000/ponies/new>, The output will be as follows

**Ruby** **on** **Rails**

C loealhosi >w:......-

**New** **Pony** **Name**

Profession

**Crea**4**a** **Pony**

&3ck

**Creating an Ajax**

Now open app/views/ponies/index.htmi.erb with suitable text editors. Update your destroy line with :remote => true, :class => 'delete\_pony'.At finally, it looks like as follows

'mdex.html.erb \*

<tn >Profession</tH> <th colspan\*"

j"x

</tr>

</th\*ad> /th

14 **<***<%*Ironies.each do |pony| %> **<tr»**

**tbody***?*

<?d><V pony.nawe %></td> «tdxX» pony.profession *%></td\**

*<tdX%»* lir»k\_to ’Show', pony %><Ad>

<tdxl« link\_to 'Cdlt\*, edltj>Oflyp«th(pony) Xx/td>

-

*<tdr<%»* lin»c\_t© ‘Destroy\*, pony, «ethod: delete, date: { confirm: ‘Are you sure?’ •> true, :cla\t •> 'delete\_pcny' *%>* </trf>

</tr>end *%>* </tbody>

*<%*

</table>

<br>

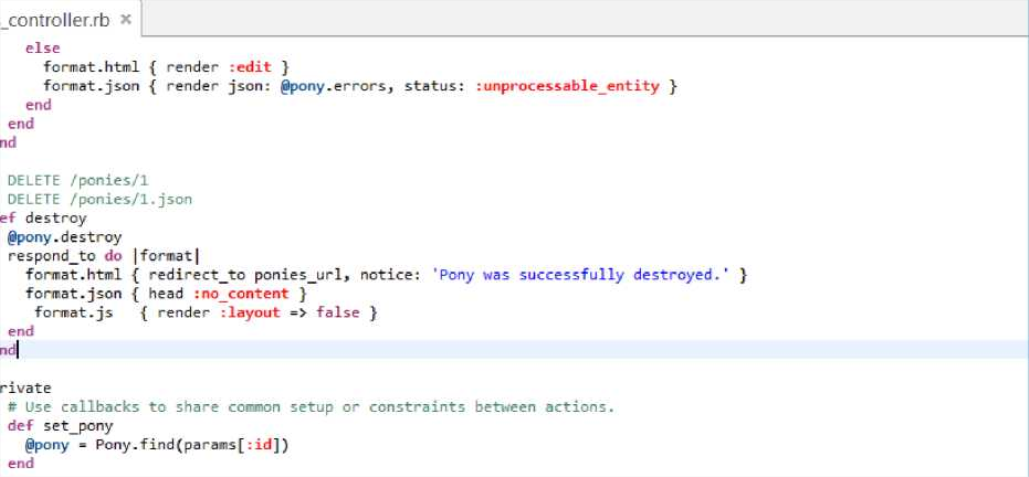
*<%•* llnk^to 'tew Pony\*, newj»ony\_path *%>*

Create a file, destroy.js.erb, put it next to your other .erb files (under app/views/ponies). It should look like this:

*A* £5? views.

|> £5” layouts *A* £5? ponies *m.* form.htm I. erb « destroy.js.erb «

edrt.html.erb « index, html. erb indeK.json.jbuilder « new.html. erb « show.htm I, erb

**Ruby** **on** **Rails**

Now enter the code as shown below in destroy .js.erb

$('.delete\_pony').bind('ajax:success', function() {

$(this).closest('tr').fadeOut(); });

Now Open your controller file which is placed at app/controllers/ponies\_controller.rb and add the following code in destroy method as shown below

# DELETE /ponies/1 # DELETE /ponies/1.json def destroy

@pony = Pony.find(params[:id])

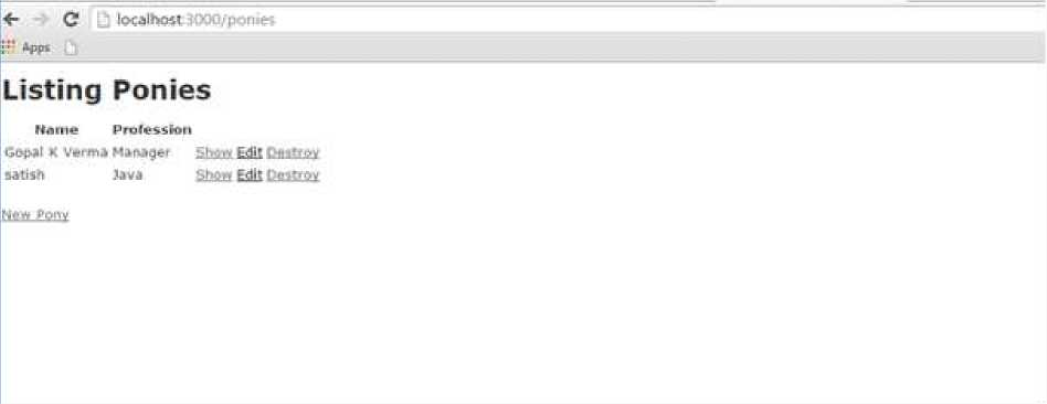
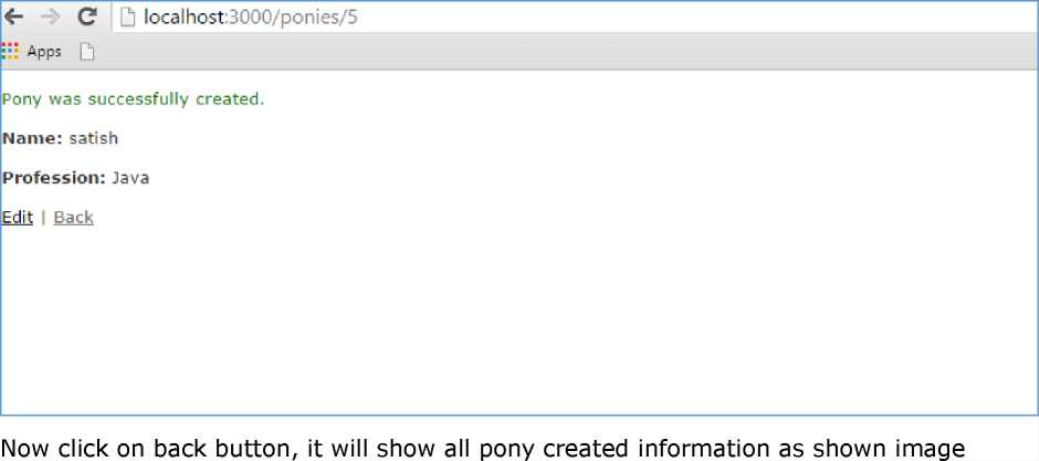
@pony.destroy

respond\_to do |format|

format.html { redirect\_to ponies\_url } format.json { head :no\_content } format.js

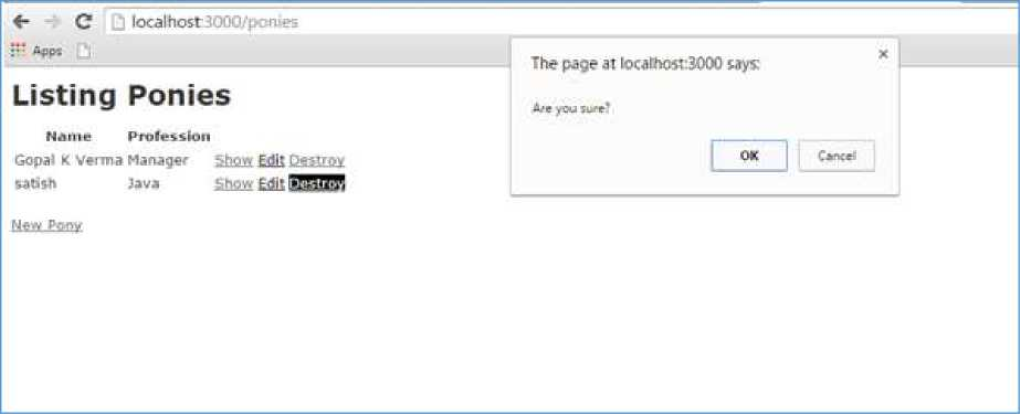
{ render :layout => false } end end At finally controller page is as shown image.

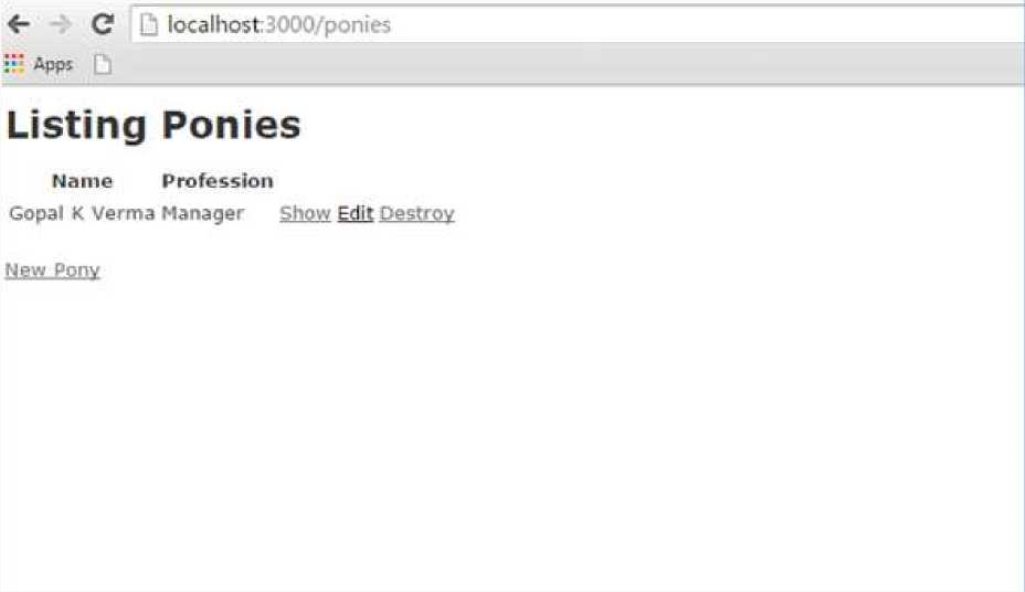
Now run an application, Output called from <http://localhost:3000/ponies/new>, it will looks like as following image

**Ruby** **on** **Rails**

Press on create pony button, it will generate the result as follows

Till now, we are working on scaffold, now click on destroy button, it will call a pop-up as shown below image, the pop-up works based on Ajax.

**Ruby** **on** **Rails**

If Click on ok button, it will delete the record from pony.Here I have clicked ok button.Final output will be as follows

**15.** **Ruby** **on** **Rails** **-** **File** **Uploading**

You may have a requirement in which you want your site visitors to upload a file on your server. Rails makes it very easy to handle this requirement. Now we will proceed with a simple and small Rails project.

As usual, let's start off with a new Rails application called **testfile**. Let's create the basic structure of the application by using simple rails command.

tp> rails new testfile

Before starting application development, we should install gem files as shown below:

gem install carrierwave

gem install bootstrap-sass

Open up your gemfile and add the following two gems at the bottom as shown in the following image:

43 ■ ^lndm4 dcri not intlutfe ioneinfo flies, so bundle the tiinfo-dala ge^ -U g\*# ! tslnfe-data\*, p-Urf\*\*1\*\*; : Jruby]

41

44> 'Cifriei^frve'\* 0.9"

41 jgMt ‘bootstrap-vats'1 »■

After adding gems in the gem file, we need to run the following command on the console:

bundle install

**Creating** **the** **Model**

We need to create a model with two strings as name and attachment as shown below:

rails g model Resume name:string attachment:string

We need to create the database migration as shown below:

rake db:migrate

We need to generate the controller as shown below:

rails g controller Resumes index new create destroy

Great! Now we have the basic structure set up. Now we need to create an uploader. An Uploader came from carrierwave gem and it tells to carrierwave how to handle the files. In short, it contained all file processing functionalities. Run the command to create an uploader as shown below

**Ruby** **on** **Rails**

rails g uploader attachment

Now open the resume model and call the uploader as shown below. Resume model has placed at app/models/resume.rb:

class Resume < ActiveRecord::Base

mount\_uploader :attachment, AttachmentUploader # Tells rails to use this uploader for this model.

validates :name, presence: true # Make sure the owner's name is present. end Before working on controller, we need to modify our config/routes.db as shown below

CarrierWaveExample::Application.routes.draw do

resources :resumes, only: [:index, :new, :create, :destroy] root

"resumes#index" end

Let us edit the controller as shown below.

class ResumesController < ApplicationController def index

@resumes = Resume.all end

def new

@resume = Resume.new end

def create

@resume = Resume.new(resume\_params) if @resume.save

redirect\_to resumes\_path, notice: "The resume #{@resume.name} has been uploaded."

else

render "new" end end

def destroy

**Ruby** **on** **Rails**

@resume = Resume.find(params[:id])

@resume.destroy

redirect\_to resumes\_path, notice: "The resume #{@resume.name} has been deleted."

end

private

def resume\_params

params.require(:resume).permit(:name, :attachment) end end

Let's add bootstrap implementation in css file.css file could be in app/assets/stylesheets/resumes.css.scss

@import "bootstrap";

Now open up app/views/layouts/application.html.erb and add codes as shown below

<!DOCTYPE html>

<html>

<head>

<title>Tutorialspoint</title>

*<%=* stylesheet\_link\_tag "application", media: "all", "data-turbolinks-track" => true *%>*

<%= javascript\_include\_tag "application", "data-turbolinks-track" => true %>

<%= csrf\_meta\_tags %>

</head>

<body>

<div class="container" style="padding-top:20px;">

<%= yield %>

</div>

</body>

</html>

Now we need to set up index views as shown below:

<% if !flash[:notice].blank? %>

<div class="alert alert-info">

<%= flash[:notice] %>

**Ruby** **on** **Rails**

</div>

*<%* end %>

<br />

<%= link\_to "New Resume", new\_resume\_path, class: "btn btn-primary" %>

<br />

<br />

<table class="table table-bordered table-striped">

<thead>

<tr>

<th>Name</th>

<th>Download Link</th>

<th> </th>

</tr>

</thead>

<tbody>

<% @resumes.each do |resume| %>

<tr>

<td><%= resume.name %></td>

<td><%= link\_to "Download Resume", resume.attachment\_url %></td> <td><%= button\_to "Delete", resume, method: :delete, class: "btn btn-

danger", confirm: "Are you sure that you wish to delete #{resume.name}?" %></td>

</tr>

<% end %>

</tbody>

</table>

Now, let us edit new.html.erb and add our form code.

<% if !@resume.errors.empty? %>

<div class="alert alert-error">

<ul>

<% @resume.errors.full\_messages.each do |msg| %>

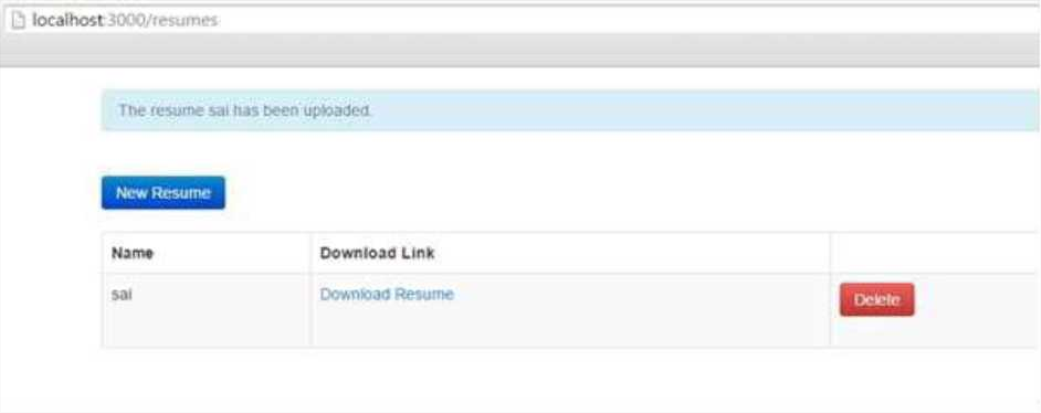
<li><%= msg %></li>

<% end %>

</ul>

</div>

<% end %>

**Ruby** **on** **Rails**

<div class="well">

*<%=* form\_for @resume, html: { multipart: true } do |f| %>

<%= f.label :name %>

<%= f.text\_field :name %>

<%= f.label :attachment %>

<%= f.file\_field :attachment %>

<%= f.submit "Save", class: "btn btn-primary" %>

<% end %>

</div>

Now start the server and visit <http://localhost:3000>. It will produce a screen similar to as follows:

One last thing we need to do is filter the list of allowed filetypes. For that we need add simple code as shown below at app/uploaders/attachment\_uploader.rb

class AttachmentUploader < CarrierWave::Uploader::Base storage file

def store\_dir

"uploads/#{model.class.to\_s.underscore}/#{mounted\_as}/#{model.id}" end

def extension\_white\_list %w(pdf doc htm html docx) end end

**Ruby** **on** **Rails**

Now start the server and visit <http://localhost:3000>. Now input a wrong format; it will generate a wrong message as shown below:

For a complete detail on **File** object, you need to go through the **Ruby** **Reference** **Manual.**

**16.** **Ruby** **on** **Rails** **-** **Send** **Email**

**Action** **Mailer** is the Rails component that enables applications to send and receive emails. In this chapter, we will see how to send an email using Rails. Let's start creating an **emails** project using the following command.

> rails emails

This will create the required framework to proceed. Now, we will start with configuring the ActionMailer.

**Action** **Mailer** **-** **Configuration**

Following are the steps you have to follow to complete your configuration before proceeding with the actual work:

Go to the config folder of your emails project and open **environment.rb** file and add the following line at the bottom of this file.

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

It tells ActionMailer that you want to use the SMTP server. You can also set it to be :sendmail if you are using a Unix-based operating system such as Mac OS X or Linux.

Add the following lines of code at the bottom of your environment.rb as well.

config.action\_mailer.smtp\_settings = {

address:

port:

domain:

user\_name:

password: authentication:

'smtp.gmail.com',

587,

'example.com',

'&rt;username&gt;1,

'&lt;password&gt;1,

'plain',

enable\_starttls\_auto: true }

Replace each hash value with proper settings for your Simple Mail Transfer Protocol (SMTP) server. You can take this information from your Internet Service Provider if you already don't know. You don't need to change port number 25 and authentication type if you are using a standard SMTP server.

You may also change the default email message format. If you prefer to send email in HTML instead of plain text format, add the following line to config/environment.rb as well:

ActionMailer::Base.default\_content\_type = "text/html"

**Ruby** **on** **Rails**

ActionMailer::Base.default\_content\_type could be set to "text/plain", "text/html", and "text/enriched". The default value is "text/plain".

The next step will be to create a mailer.

**Generate a Mailer\_**

Use the following command to generate a mailer as follows:

tp> cd emails

emails> rails generate mailer Usermailer

This will create a file user\_mailer.rb in the app\mailer directory. Check the content of this file as follows:

class Emailer < ActionMailer::Base end Let's create one method as follows:

class UserMailer < ApplicationMailer

default from: 'notifications@example.com'

def welcome\_email(user)

@user = user

@url = 'http://www.gmail.com'

mail(to: @user.email, subject: 'Welcome to My Awesome Site') end end

**•** **default** **Hash** - This is a hash of default values for any email you send from this mailer. In this case we are setting the from header to a value for all messages in this class. This can be overridden on a per-email basis.

**•** **mail** - The actual email message, we are passing the :to and :subject headers in.

Create a file called welcome\_email.html.erb in app/views/user\_mailer/. This will be the template used for the email, formatted in HTML:

<html>

<head>

<meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />

</head>

<body>

<h1>Welcome to example.com, <%= @user.name %></h1> <p>

**Ruby** **on** **Rails**

You have successfully signed up to example.com, your

username is: <%= @user.login %>.<br>

</p> <p>

To login to the site, just follow this link: <%= @url %>.

</p>

<p>Thanks for joining and have a great day!</p>

</body>

</html>

Next we will create a text part for this application as follow

Welcome to example.com, <%= @user.name %>

You have successfully signed up to example.com, your

username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!

**Calling** **the** **Mailer**

First, let's create a simple User scaffold

$ bin/rails generate scaffold user name email login $ bin/rake db:migrate

Action Mailer is nicely integrated with Active Job so you can send emails outside of the request-response cycle, so the user doesn't have to wait on it:

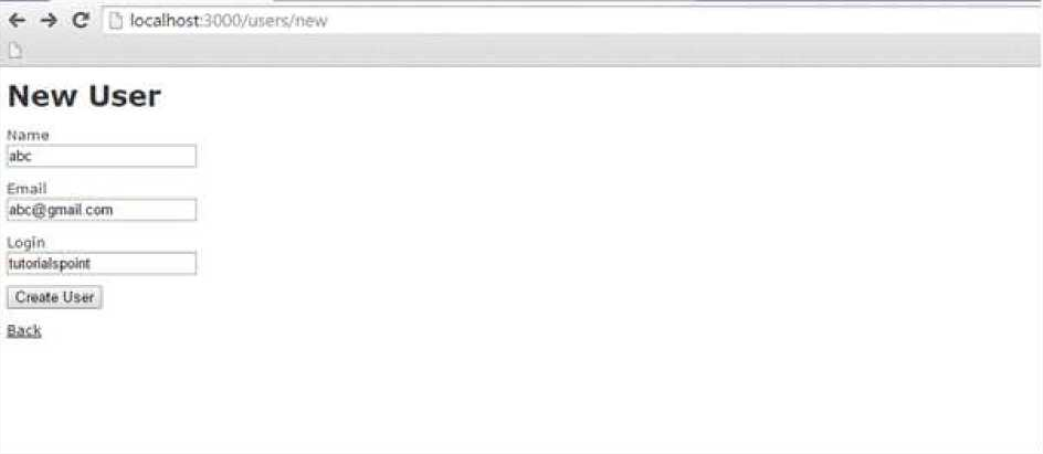
class UsersController < ApplicationController

# POST /users

# POST /users.json def create

@user = User.new(params[:user])

respond\_to do |format| if @user.save

**Ruby** **on** **Rails**

# Tell the UserMailer to send a welcome email after save

UserMailer.welcome\_email(@user).deliver\_later

format.html

{ created.') } redirect\_to(@user, notice: 'User was successfully render

format.json {

else

format.html

{

json: @user, status: :created, location: @user }

render action: 'new' }

render json: @user.errors, status: :unprocessable\_entity

format.json { } end

end

end

end

Now, test your application by using <http://127.0.0.1:3000/users/new>. It displays the following screen and by using this screen, you will be able to send your message to anybody.

This will send your message and will display the text message "Message sent successfully" and output as follow

**Ruby** **on** **Rails**