## 1. Ruby on Rails 101, or Master the Beginner Level

**Learn CSS, HTML and JavaScript**

You might wonder why CSS, HTML and JavaScript always go together. Put simply, HTML structures a website’s content (tables, blocks, images, etc.) with the help of tags. CSS then says how these website elements should be displayed. Finally, JavaScript makes a website dynamic so it can respond to clicks, create passwords or change values in form inputs.

CSS, HTML and JavaScript are the technologies you need to learn to become a front-end developer. Since Ruby on Rails is a full-stack framework that covers backend (what happens on the server) as well as front-end web development (what happens in the browser), it’s important to start with the front-end basics.

Here a few tools to help you learn the basics of front-end web development:

* The Intro to HTML and CSS course on Udacity doesn’t follow the structure of similar courses. This Udacity course gets you thinking like a real front-end developer: you won’t focus on a language’s syntax. Instead, you’ll learn how to convert design mockups to web pages step by step. What’s more, the course includes a chapter on responsive design techniques, something rarely taught within courses about HTML/CSS.
* The Make a Website course by Codecademy teaches you the fundamentals of CSS and HTML while you build your first website. The course is distributed for free, however, you can get additional features with a paid plan.

### earn Ruby Before Rails

Just to be clear, Ruby is a programming language, while Ruby on Rails (hint: you’ll sound more knowledgeable if you just say “Rails”) is a web application framework – a collection of pre-written code that simplifies website building. Rails extends the Ruby language and solves everyday problems so you don’t have to reinvent the wheel.

Beginner developers often wonder if they need to learn Ruby language first before moving to Ruby on Rails. We think you should. Our advice is to learn Ruby first. Of course, Rails offers great solutions out of the box; but when you face more serious issues, you’ll have to struggle with the Ruby code.

There are many ways to learn Ruby: online courses, books, video tutorials, and even mentorship programs. If you want to learn independently, you’ll probably dig into books and online courses, which are popular today. But remember: no matter how you learn Ruby, the way to succeed is to practice what you learn. A common mistake of beginners is gaining book knowledge without practicing: theory is great, but if you can’t put it into practice it’s not worth much.

We’ve picked out a few books and courses that’ll help you learn the basics of Ruby:

* **Code School**. Code School is an online platform where you can master Ruby and other programming languages including HTML/CSS, JavaScript, Python, and .Net. What makes Code School stand out is their entertaining approach: every lesson has a storyline which you follow while reading short explanations, watching video tutorials and practicing coding straight in the browser. Our RubyGarage team recommends taking the free Try Ruby course. In fact, Code School courses are part of our own educational program. RubyGarage has been running Ruby/Rails courses since we launched in 2011. We implement home assignments on Code School to better immerse our mentees in Ruby/Rails technologies.
* **Learn to Program (The Facets of Ruby Series) by Chris Pine.** This is the book for complete beginners at programming. From writing your first single-line program to creating a basic video game, you’re guided carefully through the learning process. Learn to Program enforces that programming can be learned only by practicing.

**Learn Ruby on Rails**

Once you know all about Ruby, you can finally start learning Rails! Many novice developers are wondering why learn Ruby on Rails if there are other frameworks. However, the answer is clear: this web application framework is a perfect solution for startups thanks to its extensive out-of-the-box functionality, deployment speed and scalability. Let’s see what educational platforms can help you learn Ruby on Rails online and what books you should read to master the basics of RoR.

In general, online courses for Rails all look the same and offer the same products. We’ve picked out a few platforms that we personally like:

* **Code School’s Rails for Zombies Redux** courses. The playful name of this course shouldn’t fool you: you’ll learn CRUD principles, Active Record models and MVC architecture. This course is designed for those who already have some knowledge of programming, web development and Ruby. By the way – after successfully completing of this course you get a badge you can put on your CV.
* **Codecademy’s** Learn Ruby on Rails course is designed for beginner programmers. Codecademy is a platform for learning skills that you can apply in real life straight after completing a course. At Codecademy you’ll learn how to build a Flipboard-like home page and how to manage versions of your code with Git among plenty of other useful skills. Codecademy lessons combine theory with an in-browser simulator to practice coding. A small tip: take notes while reading the reference material to help you retain information.

We’ve also picked up a few books that our Ruby/Rails developers love and recommend.

* **Agile Web Development with Rails 4 by Sam Ruby, David Thomas, and David Heinemeier Hansson.**This is the ultimate guide to basics including Rails installation and architecture (models, views, and controllers). You’ll also learn a bit of Ruby before you create your very first storefront following the practical section of the book. *Agile Web Development*’s complexity increases gradually throughout the book. The Practical approach of this book is supported by story-driven chapters, iterative exercises and hand-holding narration.
* **Ruby on Rails Tutorial by Michael Hartl.**Many Ruby/Rails developers admit they’ve learned Rails by simply following Michael Hartl’s Rails tutorial. The book takes you by the hand and guides you through full cycle of web development: from coding best practices to writing tests. All the chapters are tied with exercises to practice what you’ve just learned; the main project is a Twitter-like app.

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.

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>

<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**.

tp> 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>

# 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

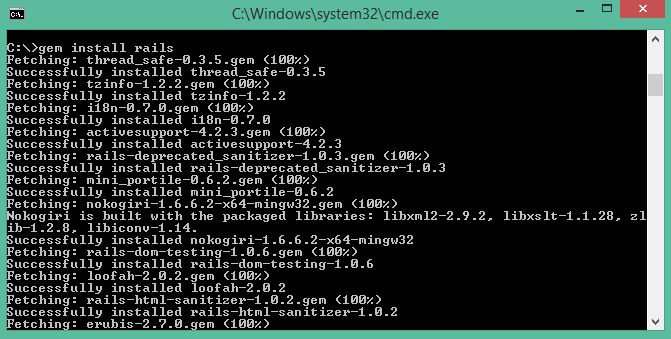
If Ruby is not installed, then download an installation package from rubyinstaller.org. Follow the **download** link, and run the resulting installer. This is an exe file **rubyinstaller-2.2.2.x.exe** and will be installed in a single click. It's a very small package, and you'll get RubyGems as well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



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.

Step 4: Check Rails Version

Use the following command to check the rails version.

C:\> rails -v

**Output**

Rails 4.2.4

Congratulations! You are now on Rails over Windows.

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

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.

Step 5: Install JavaScript Runtime

Let us 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

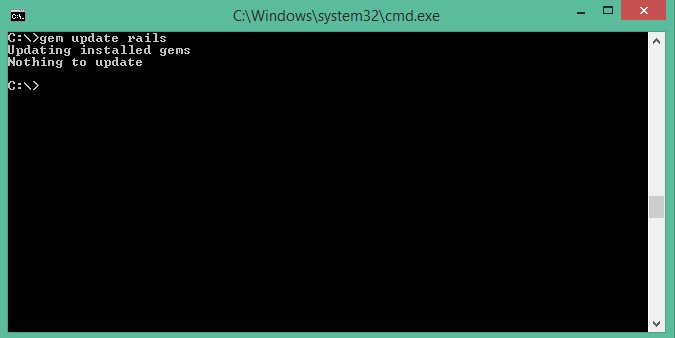
Keeping Rails Up-to-Date

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.



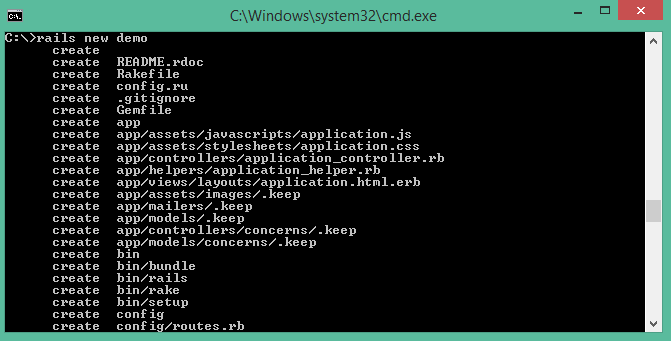
This will automatically update your Rails installation. The next time you restart your application, it will pick up this latest version of Rails. While using this command, make sure you are connected to the internet.

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**

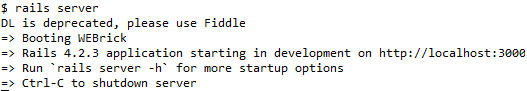


It will generate a demo rail project; we will discuss about it later. Currently we have to check if the environment is set up or not. Next, use the following command to run WEBrick web server on your machine.

tp> cd demo

tp> rails server

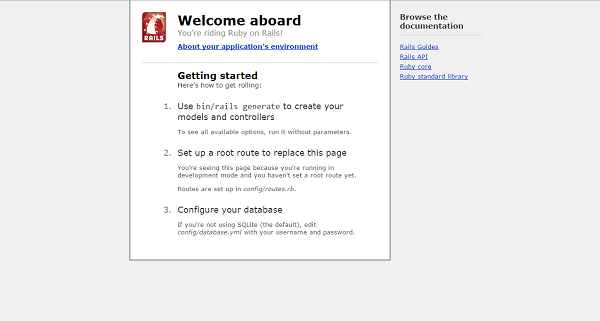
It will generate auto-code to start the server



Now open your browser and type the following −

http://localhost:3000

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



# Ruby on Rails - Framework

 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.

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.

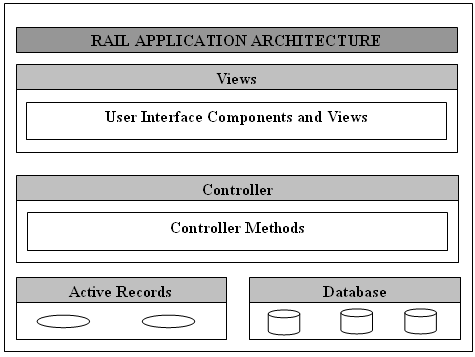
Controller ( ActionController )

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, messaging 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/2.2.0/gems

tp> ls

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

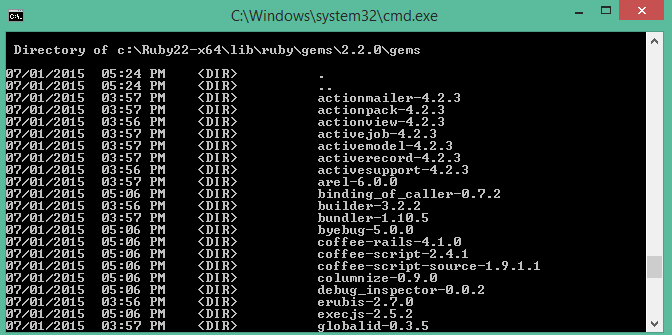
* actionpack-x.y.z
* activerecord-x.y.z
* rails-x.y.z

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

tp>cd ruby\lib\ruby\gems\2.2.0\gems

ruby\lib\ruby\gems\2.2.0\gems\>dir

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



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.

# Ruby on Rails - Directory Structure

When you use the Rails helper script to create your application, it creates the entire directory structure for the application. Rails knows where to find things it needs within this structure, so you don't have to provide any input.

Here is a top-level view of a directory tree created by the helper script at the time of application creation. Except for minor changes between releases, every Rails project will have the same structure, with the same naming conventions. This consistency gives you a tremendous advantage; you can quickly move between Rails projects without relearning the project's organization.

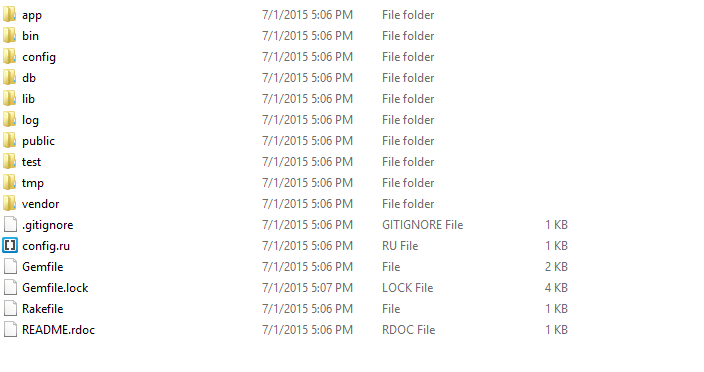
To understand this directory structure, let's use the **demo** application created in the Installation chapter. It can be created using a simple helper command **rails demo**.

Now, go into the demo application root directory as follows −

tp> cd demo

demo> dir

You will find a directory structure in Windows as follows −



Now let's explain the purpose of each directory

* **app** − It organizes your application components. It's got subdirectories that hold the view (views and helpers), controller (controllers), and the backend business logic (models).
* **app/controllers** − The controllers subdirectory is where Rails looks to find the controller classes. A controller handles a web request from the user.
* **app/helpers** − 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.
* **app/models** − 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!
* **app/view** − 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.
* **app/view/layouts** − Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views. In your views, define a layout using the <tt>layout:default</tt> and create a file named default.html.erb. Inside default.html.erb, call <% yield %> to render the view using this layout.
* **components** − This directory holds components, tiny self-contained applications that bundle model, view, and controller.
* **config** − This directory contains the small amount of configuration code that your application will need, including your database configuration (in database.yml), your Rails environment structure (environment.rb), and routing of incoming web requests (routes.rb). You can also tailor the behavior of the three Rails environments for test, development, and deployment with files found in the environments directory.
* **db** − Usually, your Rails application will have model objects that access relational database tables. You can manage the relational database with scripts you create and place in this directory.
* **doc** − Ruby has a framework, called RubyDoc, that can automatically generate documentation for code you create. You can assist RubyDoc with comments in your code. This directory holds all the RubyDoc-generated Rails and application documentation.
* **lib** − You'll put libraries here, unless they explicitly belong elsewhere (such as vendor libraries).
* **log** − Error logs go here. Rails creates scripts that help you manage various error logs. You'll find separate logs for the server (server.log) and each Rails environment (development.log, test.log, and production.log).
* **public** − Like the public directory for a web server, this directory has web files that don't change, such as JavaScript files (public/javascripts), graphics (public/images), stylesheets (public/stylesheets), and HTML files (public).
* **script** − This directory holds scripts to launch and manage the various tools that you'll use with Rails. For example, there are scripts to generate code (generate) and launch the web server (server).
* **test** − The tests you write and those that Rails creates for you, all goes here. You'll see a subdirectory for mocks (mocks), unit tests (unit), fixtures (fixtures), and functional tests (functional).
* **tmp** − Rails uses this directory to hold temporary files for intermediate processing.
* **vendor** − Libraries provided by third-party vendors (such as security libraries or database utilities beyond the basic Rails distribution) go here.

Apart from these directories, there will be two files available in demo directory.

* **README** − This file contains a basic detail about Rail Application and description of the directory structure explained above.
* **Rakefile** − This file is similar to Unix Makefile, which helps with building, packaging and testing the Rails code. This will be used by rake utility supplied along with the Ruby installation.

# Ruby on Rails - Examples

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 work flow for creating Rails Application is as follows −

* Use the rails command to create the basic skeleton of the application.
* Create a database on the PostgreSQL server to hold your data.
* Configure the application to know where your database is located and the login credentials for it.
* Create Rails Active Records (Models), because they are the business objects you'll be working with in your controllers.
* Generate Migrations that simplify the creating and maintaining of database tables and columns.
* Write Controller Code to put a life in your application.
* 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. It will create the directory structure in the current directory.

tp> rails new 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.

Most of our development work will be creating and editing files in the **library/app** subdirectories. Here's a quick run down 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

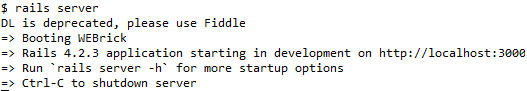
Rails web application can run under virtually any web server, but the most convenient way to develop a Rails web application is to use the built-in WEBrick web server. Let's start this web server and then browse to our empty library application −

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

tp> cd ruby\library

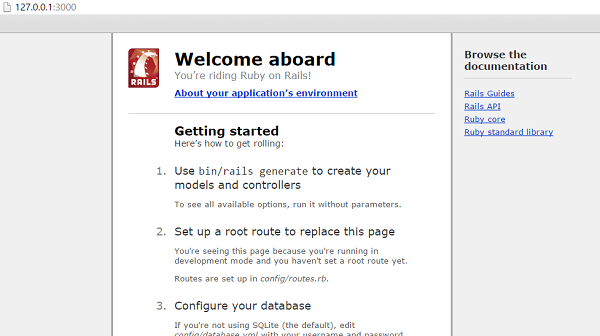
tp\ruby\library\> Rails server

It generates the auto code to start the server as shown below −



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. If everything goes well it will generate the output as follows.



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 - Database Setup

Before starting with this chapter, make sure your database server is up and running. Ruby on Rails recommends to create three databases - a database each for development, testing, and production environment. According to convention, their names should be −

* library\_development
* library\_production
* library\_test

You should initialize all three of them and create a user and password for them with full read and write privileges. We are using the **root** user ID for our application.

Database Setup for MySQL

In MySQL, we are using the **root** user ID for our application. The MySQL console session in which you do this looks something like −

mysql> create database library\_development;

Query OK, 1 row affected (0.01 sec)

mysql> grant all privileges on library\_development.\*

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)

You can do the same thing for two more databases **library\_production** and **library\_test**.

Configuring database.yml

At this point, you need to let Rails know about the user name and password for the databases. You do this in the file **database.yml**, available in the **library\config** subdirectory of Rails Application you created. This file has live configuration sections for MySQL databases. In each of the sections you use, you need to change the username and password lines to reflect the permissions on the databases you've created.

When you finish, it should look something like −

development:

adapter: mysql

database: library\_development

username: root

password: [password]

host: localhost

test:

adapter: mysql

database: library\_test

username: root

password: [password]

host: localhost

production:

adapter: mysql

database: library\_production

username: root

password: [password]

host: localhost

Database Setup for PostgreSQL

By default, PostgreSQL does not provide any users. We have to create new users. Use the following command to create a user with the name **rubyuser**.

tp> sudo -u postgres createuser rubyuser -s

If you want to create a password for the new user, then use the following command.

tp> sudo -u postgres psql

postgres=# \password rubyuser

Use the following command for creating a database **library\_development**.

postgres=# CREATE DATABASE library\_development OWNER rubyuser;

CREATE DATABASE

Use the following command for creating a database **library\_production**.

postgres=# CREATE DATABASE library\_production OWNER rubyuser;

CREATE DATABASE

Use the following command for creating a database **library\_test**.

postgres=# CREATE DATABASE library\_test OWNER rubyuser;

CREATE DATABASE

Press **Ctrl+D** to terminate PosgreSQL.

Configuring database.yml

At this point, you need to let Rails know the username and password for the databases. You do this in the file **database.yml**, available in the **library\config** subdirectory of Rails Application you created. This file has live configuration sections for PostgreSQL databases. In each of the sections, you need to change the username and password lines to reflect the permissions on the databases you've created.

When you finish, it should look as follows −

default: &default

adapter: postgresql

encoding: unicode

development:

adapter: postgresql

encoding: unicode

database: library\_development

username: rubyuser

password: <Password for rubyuser>

test:

adapter: postgresql

encoding: unicode

database: library\_test

username: rubyuser

password: <Password for rubyuser>

production:

adapter: postgresql

encoding: unicode

database: library\_production

username: rubyuser

password: <Password for rubyuser>

What is Next?

The next two chapters explain how to model your database tables and how to manage those using Rails Migrations.

# Ruby on Rails - Active Records

Rails Active Record is the Object/Relational Mapping (ORM) layer supplied with Rails. It closely follows the standard ORM model, which is as follows −

* tables map to classes,
* rows map to objects and
* columns map to object attributes.

Rails Active Records provide 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.

Each Active Record object has CRUD (**C**reate, **R**ead, **U**pdate, and **D**elete) methods for database access. This strategy allows simple designs and straight forward mappings between database tables and application objects.

Translating a Domain Model into SQL

Translating a domain model into SQL is generally straight forward, as long as you remember that you have to write Rails-friendly SQL. In practical terms, you have to follow certain rules −

* Each entity (such as book) gets a table in the database named after it, but in the plural (books).
* Each such entity-matching table has a field called *id*, which contains a unique integer for each record inserted into the table.
* Given entity x and entity y, if entity y belongs to entity x, then table y has a field called x\_id.
* The bulk of the fields in any table store the values for that entity's simple properties (anything that's a number or a string).

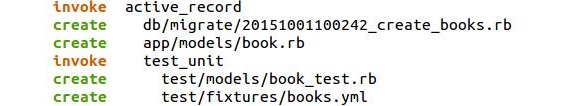
Creating Active Record Files (Models)

To create the Active Record files for our entities for library application, introduced in the previous chapter, issue the following command from the top level of the application directory.

library\> rails script/generate model Book

library\> rails script/generate model Subject

Above **rails generate model book** commands generates the auto code as below −



You're telling the generator to create models called Book and Subject to store instances of books and subjects. Notice that you are capitalizing Book and Subject and using the singular form. This is a Rails paradigm that you should follow each time you create a model.

When you use the generate tool, Rails creates the actual model file that holds all the methods unique to the model and the business rules you define, a unit test file for performing test-driven development, a sample data file (called fixtures) to use with the unit tests, and a Rails migration that makes creating database tables and columns easy.

Apart from creating many other files and directories, this will create files named **book.rb** and **subject.rb** containing a skeleton definition in the **app/models** directory.

Content available in book.rb −

class Book < ActiveRecord::Base

end

Content available in subject.rb −

class Subject < ActiveRecord::Base

end

Creating Associations between Models

When you have more than one model in your rails application, you would need to create connection between those models. You can do this via associations. Active Record supports three types of associations −

* **one-to-one** − A one-to-one relationship exists when one item has exactly one of another item. For example, a person has exactly one birthday or a dog has exactly one owner.
* **one-to-many** − A one-to-many relationship exists when a single object can be a member of many other objects. For instance, one subject can have many books.
* **many-to-many** − A many-to-many relationship exists when the first object is related to one or more of a second object, and the second object is related to one or many of the first object.

You indicate these associations by adding declarations to your models: has\_one, has\_many, belongs\_to, and has\_and\_belongs\_to\_many.

Now, you need to tell Rails what relationships you want to establish within the library data system. To do so, modify book.rb and subject.rb to look like this −

class Book < ActiveRecord::Base

belongs\_to :subject

end

We have used a singular subject in the above example, because one Book can belong to a single Subject.

class Subject < ActiveRecord::Base

has\_many :books

end

We have used plural books here, because one subject can have multiple books.

Implementing Validations on Models

The implementation of validations is done in a Rails model. The data you are entering into the database is defined in the actual Rails model, so it only makes sense to define what valid data entails in the same location.

The validations are −

* The value of title field should not be NULL.
* The value of price field should be numeric.

Open **book.rb** in the **app\model** subdiractory and put the following validations −

class Book < ActiveRecord::Base

belongs\_to :subject

validates\_presence\_of :title

validates\_numericality\_of :price, :message=>"Error Message"

end

* **validates\_presence\_of** − protects "NOT NULL" fields against missing user input.
* **validates\_numericality\_of** − prevents the user, entering non numeric data.

Besides the validations mentioned above, there are other common validations. Check Rails Quick Guide.

What is Next?

In the next chapter, we will learn Rails Migration, which allows you to use Ruby to define changes to your database schema, making it possible to use a version control system to keep things synchronized with the actual code.

# Ruby on Rails - Migrations

Rails Migration allows you to use Ruby to define changes to your database schema, making it possible to use a version control system to keep things synchronized with the actual code.

This has many uses, including −

* **Teams of developers** − If one person makes a schema change, the other developers just need to update, and run "rake migrate".
* **Production servers** − Run "rake migrate" when you roll out a new release to bring the database up to date as well.
* **Multiple machines** − If you develop on both a desktop and a laptop, or in more than one location, migrations can help you keep them all synchronized.

What Can Rails Migration Do?

* create\_table(name, options)
* drop\_table(name)
* rename\_table(old\_name, new\_name)
* add\_column(table\_name, column\_name, type, options)
* rename\_column(table\_name, column\_name, new\_column\_name)
* change\_column(table\_name, column\_name, type, options)
* remove\_column(table\_name, column\_name)
* add\_index(table\_name, column\_name, index\_type)
* remove\_index(table\_name, column\_name)

**Migrations support all the basic data types** − The following is the list of data types that migration supports −

* **string** − for small data types such as a title.
* **text** − for longer pieces of textual data, such as the description.
* **integer** − for whole numbers.
* **float** − for decimals.
* **datetime and timestamp** − store the date and time into a column.
* **date and time** − store either the date only or time only.
* **binary** − for storing data such as images, audio, or movies.
* **Boolean** − for storing true or false values.

**Valid column options are** − The following is the list of valid column options.

* **limit** ( :limit => “50” )
* **default** (:default => “blah” )
* **null** (:null => false implies NOT NULL)

**NOTE** − The activities done by Rails Migration can be done using any front-end GUI or directly on SQL prompt, but Rails Migration makes all those activities very easy.

See the Rails API for details on these.

Create the Migrations

Here is the generic syntax for creating a migration −

application\_dir> rails generate migration table\_name

This will create the file db/migrate/001\_table\_name.rb. A migration file contains the basic Ruby syntax that describes the data structure of a database table.

**NOTE** − Before running the migration generator, it is recommended to clean the existing migrations generated by model generators.

We will create two migrations corresponding to our three tables − ***books and subjects***.

Books migration should be as follows −

tp> cd library

library> rails generate migration books

Above command generates the following code.

Generate

subject migration should be as follows −

tp> cd library

library> rails generate migration subjects

Above command generates the following code.

Generate

Notice that you are using lower case for book and subject and plural form while creating migrations. This is a Rails paradigm that you should follow each time you create a Migration.

Edit the Code

Go to db/migrate subdirectory of your application and edit each file one by one using any simple text editor.

Modify 001\_books.rb as follows −

The ID column will be created automatically, so don't do it here as well.

class Books < ActiveRecord::Migration

def self.up

create\_table :books do |t|

t.column :title, :string, :limit => 32, :null => false

t.column :price, :float

t.column :subject\_id, :integer

t.column :description, :text

t.column :created\_at, :timestamp

end

end

def self.down

drop\_table :books

end

end

The method **self.up** is used when migrating to a new version, **self.down** is used to roll back any changes if needed. At this moment, the above script will be used to create ***books*** table.

Modify 002\_subjects.rb as follows −

class Subjects < ActiveRecord::Migration

def self.up

create\_table :subjects do |t|

t.column :name, :string

end

Subject.create :name => "Physics"

Subject.create :name => "Mathematics"

Subject.create :name => "Chemistry"

Subject.create :name => "Psychology"

Subject.create :name => "Geography"

end

def self.down

drop\_table :subjects

end

end

The above script will be used to create ***subjects*** table and will create five records in the subjects table.

Run the Migration

Now that you have created all the required migration files. It is time to execute them against the database. To do this, go to a command prompt and go to the library directory in which the application is located, and then type **rake migrate** as follows −

library> rake db:migrate

This will create a "schema\_info" table if it doesn't exist, which tracks the current version of the database - each new migration will be a new version, and any new migrations will be run until your database is at the current version.

**Rake** is a Ruby build program similar to Unix *make* program that Rails takes advantage of, to simplify the execution of complex tasks such as updating a database's structure etc.

Running Migrations for Production and Test Databases

If you would like to specify what Rails environment to use for the migration, use the RAILS\_ENV shell variable.

For example −

library> export RAILS\_ENV = production

library> rake db:migrate

library> export RAILS\_ENV = test

library> rake db:migrate

library> export RAILS\_ENV = development

library> rake db:migrate

**NOTE** − In Windows, use "set RAILS\_ENV = production" instead of *export*command.

What is Next?

Now we have our database and the required tables available. In the two subsequent chapters, we will explore two important components called Controller (ActionController) and View (ActionView).

* Creating Controllers (Action Controller).
* Creating Views (Action View).

# Ruby on Rails - Controller

he Rails controller is the logical center of your application. It coordinates the interaction between the user, the views, and the model. The controller is also a home to a number of important ancillary services.

* It is responsible for routing external requests to internal actions. It handles people-friendly URLs extremely well.
* It manages caching, which can give applications orders-of-magnitude performance boosts.
* It manages helper modules, which extend the capabilities of the view templates without bulking up their code.
* It manages sessions, giving users the impression of an ongoing interaction with our applications.

The process for creating a controller is very easy, and it's similar to the process we've already used for creating a model. We will create just one controller here −

library\> rails generate controller Book

Notice that you are capitalizing Book and using the singular form. This is a Rails paradigm that you should follow each time you create a controller.

This command accomplishes several tasks, of which the following are relevant here −

* It creates a file called **app/controllers/book\_controller.rb**

If you look at book\_controller.rb, you will find it as follows −

class BookController < ApplicationController

end

Controller classes inherit from *ApplicationController,* which is the other file in the controllers folder: **application.rb**.

The *ApplicationController* contains code that can be run in all your controllers and it inherits from Rails *ActionController::Base* class.

You don't need to worry with the *ApplicationController* as of yet, so let's just define a few method stubs in **book\_controller.rb**. Based on your requirement, you could define any number of functions in this file.

Modify the file to look like the following and save your changes. Note that it is upto you what name you want to give to these methods, but better to give relevant names.

class BookController < ApplicationController

def list

end

def show

end

def new

end

def create

end

def edit

end

def update

end

def delete

end

end

Now let us implement all the methods one by one.

## Implementing the list Method

The list method gives you a list of all the books in the database. This functionality will be achieved by the following lines of code. Edit the following lines in book\_controller.rb file.

def list

@books = Book.all

end

The *@books = Book.all* line in the list method tells Rails to search the books table and store each row it finds in the @books instance object.

## Implementing the show Method

The show method displays only further details on a single book. This functionality will be achieved by the following lines of code.

def show

@book = Book.find(params[:id])

end

The show method's *@book = Book.find(params[:id])* line tells Rails to find only the book that has the id defined in params[:id].

The params object is a container that enables you to pass values between method calls. For example, when you're on the page called by the list method, you can click a link for a specific book, and it passes the id of that book via the params object so that show can find the specific book.

## Implementing the new Method

The new method lets Rails know that you will create a new object. So just add the following code in this method.

def new

@book = Book.new

@subjects = Subject.all

end

The above method will be called when you will display a page to the user to take user input. Here second line grabs all the subjects from the database and puts them in an array called @subjects.

## Implementing the create Method

Once you take user input using HTML form, it is time to create a record into the database. To achieve this, edit the create method in the book\_controller.rb to match the following −

def create

@book = Book.new(book\_params)

if @book.save

redirect\_to :action => 'list'

else

@subjects = Subject.all

render :action => 'new'

end

end

def book\_params

params.require(:books).permit(:title, :price, :subject\_id, :description)

end

The first line creates a new instance variable called @book that holds a Book object built from the data, the user submitted. The **book\_params** method is used to collect all the fields from object **:books**. The data was passed from the new method to create using the params object.

The next line is a conditional statement that redirects the user to the **list**method if the object saves correctly to the database. If it doesn't save, the user is sent back to the new method. The redirect\_to method is similar to performing a meta refresh on a web page: it automatically forwards you to your destination without any user interaction.

Then *@subjects = Subject.all* is required in case it does not save data successfully and it becomes similar case as with new option.

## Implementing the edit Method

The edit method looks nearly identical to the show method. Both methods are used to retrieve a single object based on its id and display it on a page. The only difference is that the show method is not editable.

def edit

@book = Book.find(params[:id])

@subjects = Subject.all

end

This method will be called to display data on the screen to be modified by the user. The second line grabs all the subjects from the database and puts them in an array called @subjects.

## Implementing the update Method

This method will be called after the edit method, when the user modifies a data and wants to update the changes into the database. The update method is similar to the create method and will be used to update existing books in the database.

def update

@book = Book.find(params[:id])

if @book.update\_attributes(book\_param)

redirect\_to :action => 'show', :id => @book

else

@subjects = Subject.all

render :action => 'edit'

end

end

def book\_param

params.require(:book).permit(:title, :price, :subject\_id, :description)

end

The update\_attributes method is similar to the save method used by create but instead of creating a new row in the database, it overwrites the attributes of the existing row.

Then *@subjects = Subject.all* line is required in case it does not save the data successfully, then it becomes similar to edit option.

## Implementing the delete Method

If you want to delete a record from the database then you will use this method. Implement this method as follows.

def delete

Book.find(params[:id]).destroy

redirect\_to :action => 'list'

end

The first line finds the classified based on the parameter passed via the params object and then deletes it using the destroy method. The second line redirects the user to the list method using a redirect\_to call.

## Additional Methods to Display Subjects

Assume you want to give a facility to your users to browse all the books based on a given subject. So, you can create a method inside book\_controller.rb to display all the subjects. Assume the method name is **show\_subjects** −

def show\_subjects

@subject = Subject.find(params[:id])

end

Finally your **book\_controller.rb** file will look as follows −

class BooksController < ApplicationController

def list

@books = Book.all

end

def show

@book = Book.find(params[:id])

end

def new

@book = Book.new

@subjects = Subject.all

end

def book\_params

params.require(:books).permit(:title, :price, :subject\_id, :description)

end

def create

@book = Book.new(book\_params)

if @book.save

redirect\_to :action => 'list'

else

@subjects = Subject.all

render :action => 'new'

end

end

def edit

@book = Book.find(params[:id])

@subjects = Subject.all

end

def book\_param

params.require(:book).permit(:title, :price, :subject\_id, :description)

end

def update

@book = Book.find(params[:id])

if @book.update\_attributes(book\_param)

redirect\_to :action => 'show', :id => @book

else

@subjects = Subject.all

render :action => 'edit'

end

end

def delete

Book.find(params[:id]).destroy

redirect\_to :action => 'list'

end

def show\_subjects

@subject = Subject.find(params[:id])

end

end

Now save your controller file.

## What is Next?

You have created almost all the methods, which will work on backend. Next we will define routes (URLs) for actions.

# Ruby on Rails - Routes

The routing module provides URL rewriting in native Ruby. It's a way to redirect incoming requests to controllers and actions. It replaces the mod\_rewrite rules. Best of all, Rails' Routing works with any web server. Routes are defined in app/config/routes.rb.

Think of creating routes as drawing a map for your requests. The map tells them where to go based on some predefined pattern −

Rails.application.routes.draw do

Pattern 1 tells some request to go to one place

Pattern 2 tell them to go to another

...

end

## Example

Let us consider our library management application contains a controller called BookController. We have to define the routes for those actions which are defined as methods in the BookController class.

Open routes.rb file in library/config/ directory and edit it with the following content.

Rails.application.routes.draw do

get 'book/list'

get 'book/new'

post 'book/create'

patch 'book/update'

get 'book/list'

get 'book/show'

get 'book/edit'

get 'book/delete'

get 'book/update'

get 'book/show\_subjects'

end

The routes.rb file defines the actions available in the applications and the type of action such as get, post, and patch.

Use the following command to list all your defined routes, which are useful for tracking down routing problems in your application, or giving you a good overview of the URLs in an application you're trying to get familiar with.

library> rake routes

## What is Next?

Next, we will create the code to generate screens to display data and to take input from the user.

# Ruby on Rails - Views

A Rails View is an ERb program that shares data with controllers through mutually accessible variables.

If you look in the app/views directory of the library application, you will see one subdirectory for each of the controllers, we have created: book. Each of these subdirectories was created automatically when the same-named controller was created with the generate script.

Rails let's you know that you need to create the view file for each new method. Each method you define in the controller needs to have a corresponding **erb** file, with the same name as the method, to display the data that the method is collecting.

So let's create view files for all the methods we have defined in the book\_controller.rb. While executing these views, simultaneously check these actions are applicable into the database or not.

Creating View File for list Method

Create a file called **list.html.erb** using your favourite text editor and save it to app/views/book. After creating and saving the file, refresh your web browser. You should see a blank page; if you don't, check the spelling of your file and make sure that it is exactly the same as your controller's method.

Now, display the actual content. Let us put the following code into list.html.erb.

<% if @books.blank? %>

<p>There are not any books currently in the system.</p>

<% else %>

<p>These are the current books in our system</p>

<ul id = "books">

<% @books.each do |c| %>

<li><%= link\_to c.title, {:action => 'show', :id => c.id} -%></li>

<% end %>

</ul>

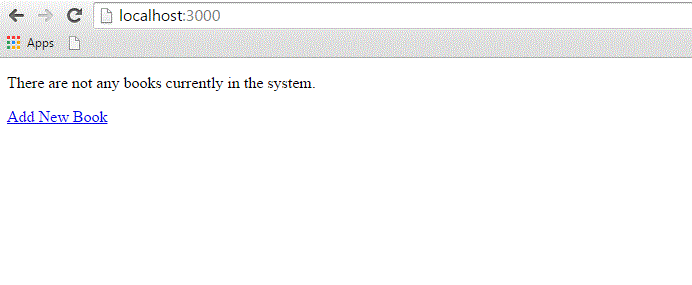
<% end %>

<p><%= link\_to "Add new Book", {:action => 'new' }%></p>

The code to be executed is to check whether the @books array has any objects in it. The **.blank?** method returns true if the array is empty, and false if it contains any objects. This @books object was created in controller inside the list method.

The code between the <%= %> tags is a **link\_to** method call. The first parameter of link\_to is the text to be displayed between the <a> tags. The second parameter is what action is called when the link is clicked. In this case, it is the show method. The final parameter is the id of the book that is passed via the params object.

Now, try refreshing your browser and you should get the following screen because we don't have any book in our library.



Creating View File for new Method

Till now, we don't have any book in our library. We have to create few books in the system. So, let us design a view corresponding to the **new** method defined in the book\_controller.rb.

Create a file called new.html.erb using your favorite text editor and save it to app/views/book. Add the following code to the new.html.erb file.

<h1>Add new book</h1>

<%= form\_tag :action => 'create' do %>

<p><label for = "book\_title">Title</label>:

<%= text\_field 'books', 'title' %></p>

<p><label for = "book\_price">Price</label>:

<%= text\_field 'books', 'price' %></p>

<p><label for = "book\_subject\_id">Subject</label>:

<%= collection\_select(:books, :subject\_id, @subjects, :id, :name, prompt: true) %></p>

<p><label for = "book\_description">Description</label><br/>

<%= text\_area 'books', 'description' %></p>

<%= submit\_tag "Create" %>

<% end -%>

<%= link\_to 'Back', {:action => 'list'} %>

Here **form\_tag** method interprets the Ruby code into a regular HTML <form> tag using all the information supplied to it. This tag, for example, outputs the following HTML −

<form action = "/book/create" method = "post">

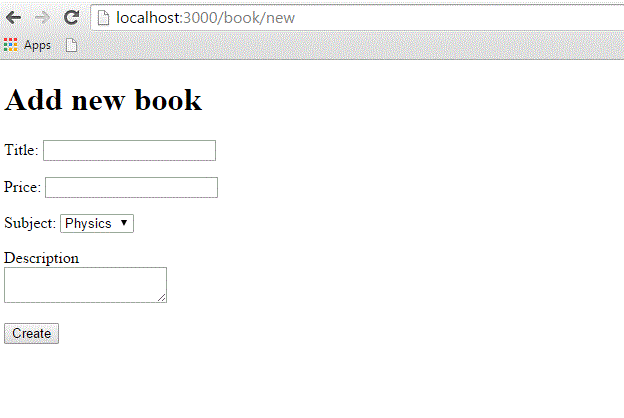
Next method is **text\_field** that outputs an <input> text field. The parameters for text\_field are object and field name. In this case, the object is *book* and the name is *title*.

Rails method called **collection\_select**, creates an HTML select menu built from an array, such as the @books one. There are five parameters, which are as follows −

* **:book** − The object you are manipulating. In this case, it's a book object.
* **:subject\_id** − The field that is populated when the book is saved.
* **@books** − The array you are working with.
* **:id** − The value that is stored in the database. In terms of HTML, this is the <option> tag's value parameter.
* **:name** − The output that the user sees in the pull-down menu. This is the value between the <option> tags.

The next used is **submit\_tag**, which outputs an <input> button that submits the form. Finally, there is the **end** method that simply translates into </form>.

Go to your browser and visit **http://localhost:3000/book/new.** This will give you the following screen.



Enter some data in this form and then click the Create button. Here i have added the following details into the fields −

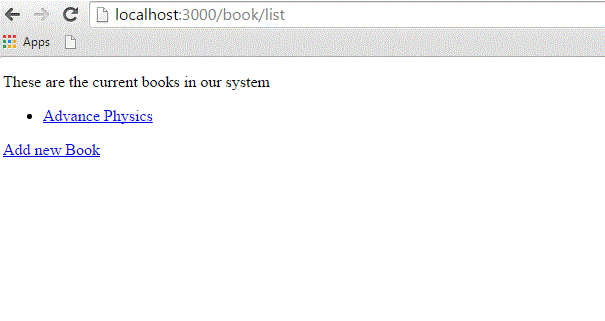
Title: Advance Physics

Price: 390

Subject: Physics

Description: This is test to create new book

When you click the **Create** button, it will call the **create** method, which does not need any view because this method is using either **list** or **new** methods to view the results. So, when you click the Create button, the data should submit successfully and redirect you to the list page, in which you now have a single item listed as follows −



If you click the link, you should see another Template is missing error, since you haven't created the template file for show method yet.

Creating View File for show Method

This method will display the complete detail about any book available in the library. Create a show.html.erb file under app/views/book and populate it with the following code −

<h1><%= @book.title %></h1>

<p>

<strong>Price: </strong> $<%= @book.price %><br />

<strong>Subject :</strong> <%= @book.subject.name %><br />

<strong>Created Date:</strong> <%= @book.created\_at %><br />

</p>

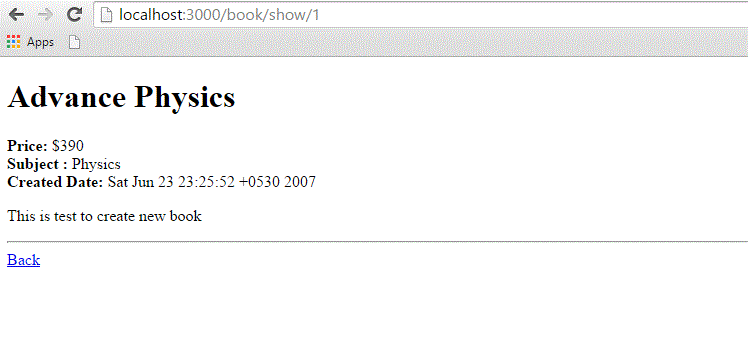
<p><%= @book.description %></p>

<hr />

<%= link\_to 'Back', {:action => 'list'} %>

This is the first time you have taken the full advantage of associations, which enable you to easily pull data from related objects.

The format used is **@variable.relatedObject.column**. In this instance, you can pull the subject's name value through the @book variable using the **belongs\_to** associations. If click on any listed record then it will show you the following screen.



Creating View File for edit Method

Create a new file called edit.html.erb and save it in app/views/book. Populate it with the following code −

<h1>Edit Book Detail</h1>

<%= form\_for @book, :url =>{:action => "update", :id =>@book} do |f| %>

<p>Title: <%= f.text\_field 'title' %></p>

<p>Price: <%= f.text\_field 'price' %></p>

<p>Subject: <%= f.collection\_select :subject\_id, Subject.all, :id, :name %></p>

<p>Description<br/>

<%= f.text\_area 'description' %></p>

<%= f.submit "Save changes" %>

<% end %>

<%= link\_to 'Back', {:action => 'list' } %>

This code is very similar to the **new** method except action to be updated instead of creating and defining an id.

In this scenario, we used **form\_for** tag for the form action. It will perform better than **form\_tag**. Why because it will create interaction with the Model easily. Therefore it is better to use form\_for tag whenever you need interaction between the model and the form fields.

At this point, we need some modification in the **list method's** view file. Go to the <li></li> element and modify it to look like the following −

<li>

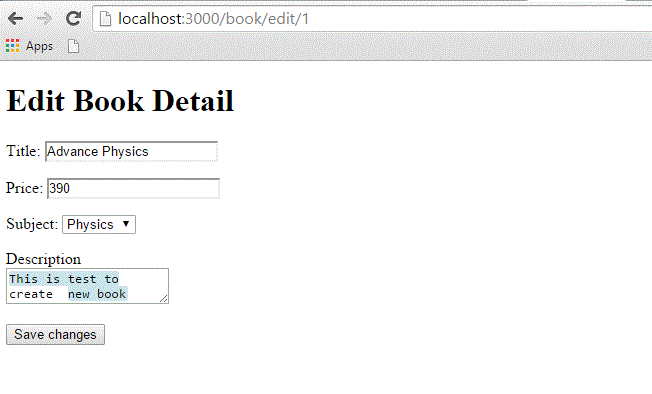
<%= link\_to c.title, {:action => "show", :id => c.id} -%>

<b> <%= link\_to 'Edit', {:action => "edit",

:id => c.id} %></b>

</li>

Now, try to browse books using the **http://localhost:3000/book/list.** It will give you the listing of all the books along with **Edit** option. When you click the Edit option, then you will have next screen as follows −



Now, you edit this information and then click the *Save Changes* button. This will result in a call to **update** method available in the controller file and it will update all the changed attribute. Notice that the **update** method does not need any view file because it's using either **show** or **edit** methods to show its results.

Creating View File for delete Method

Removing information from a database using Ruby on Rails is almost too easy. You do not need to write any view code for the delete method because this method is using **list** method to display the result. So, let's just modify list.html.erb again and add a delete link.

Go to the <li></li> element and modify it to look like the following −

<li>

<%= link\_to c.title, {:action => 'show', :id => c.id} -%>

<b> <%= link\_to 'Edit', {:action => 'edit', :id => c.id} %></b>

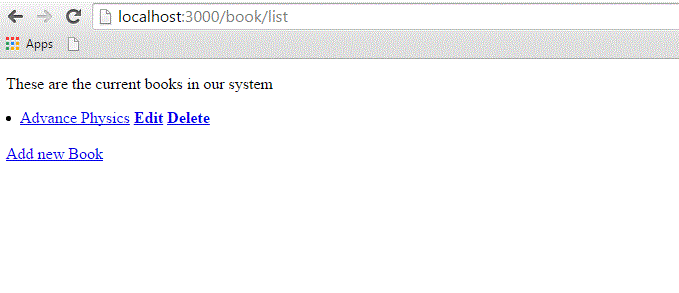
<b> <%= link\_to "Delete", {:action => 'delete', :id => c.id},

:confirm => "Are you sure you want to delete this item?" %></b>

</li>

The **:confirm** parameter presents a JavaScript confirmation box asking if you really want to perform the action. If the user clicks OK, the action proceeds, and the item is deleted.

Now, try browsing books using **http://localhost:3000/book/list.** It will give you listing of all the books along with **Edit** and **Delete** options as follows −



Now using the Delete option, you can delete any listed record.

Creating View File for show\_subjects Method

Create a new file, show\_subjects.html.erb, in the app/views/book directory and add the following code to it −

<h1><%= @subject.name -%></h1>

<ul>

<% @subject.books.each do |c| %>

<li><%= link\_to c.title, :action => "show", :id => c.id -%></li>

<% end %>

</ul>

You are taking advantage of associations by iterating through a single subject's many books listings.

Now modify the Subject: line of show.html.erb so that the subject listing shows a link.

<strong>Subject: </strong> <%= link\_to @book.subject.name,

:action => "show\_subjects", :id => @book.subject.id %><br />

This will output a list of subject on the index page, so that users can access them directly.

Modify **list.html.erb** to add the following to the top of the file −

<ul id = "subjects">

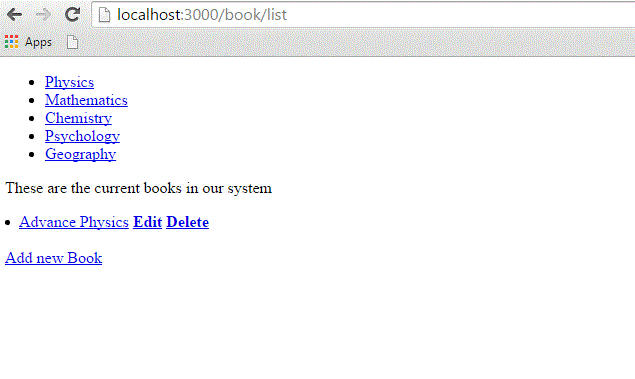
<% Subject.find(:all).each do |c| %>

<li><%= link\_to c.name, :action => "show\_subjects", :id => c.id %></li>

<% end %>

</ul>

Now try browsing books using http://localhost:3000/book/list. It will display all subjects with links so that you can browse all the books related to that subject.



What is Next?

Hope now you are feeling comfortable with all the operations of Rails.

The next chapter explains how to use **Layouts** to put your data in a better way. We will show you how to use CSS in your Rails applications.