



Cool projects that will push your skills to the limit

Rails 4 Application Development

Build simple to advanced applications in Rails 4 through 10 exciting projects

HOTSHOT

Saurabh Bhatia

www.it-ebooks.info

[PACKT] open source[®]
PUBLISHING community experience distilled

Rails 4 Application Development HOTSHOT



Build simple to advanced applications in Rails 4
through 10 exciting projects

Saurabh Bhatia

[**PACKT**]
PUBLISHING open source 
community experience distilled

BIRMINGHAM - MUMBAI

Rails 4 Application Development HOTSHOT

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2014

Production Reference: 1030414

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-629-4

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author

Saurabh Bhatia

Reviewers

Gabriel Hilal

Andrew Montgomery-Hurrell

Peter F. Philips

Philip De Smedt

Acquisition Editors

Nikhil Chinnari

Rubal Kaur

Content Development Editor

Priya Singh

Technical Editors

Venu Manthena

Mrunmayee Patil

Shruti Rawool

Copy Editors

Alisha Aranha

Mradula Hegde

Gladson Monteiro

Alfida Paiva

Project Coordinator

Leena Purkait

Proofreaders

Simran Bhogal

Maria Gould

Paul Hindle

Indexers

Rekha Nair

Priya Subramani

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Saurabh Bhatia has been developing professional software since 2005. However, his programming interests date back to his school days. Starting with Java, he quickly moved to Ruby on Rails in 2006, and it has been his primary choice of development framework since then. He built a Ruby on Rails consulting company and ran it for five years. He has worked with several companies in the tech industry, from getting two-person startups off the ground to developing software for large corporates. He is currently the CTO of Ruling Digital Inc., a software company that develops software for universities.

He has been an open source enthusiast and has helped Ubuntu penetrate the Indian market since 2007. He was a part of the open source promotion society called Twincling Society for Open Source in Hyderabad. He started and moderated Bangalore Ruby Users Group and also moderates the Mumbai Ruby Users Group. He is also a part of the RailsBridge initiative for mentoring new Rails developers.

Over the years, he has written several articles online and in print for different publications, such as *Linux User and Developer*, *Linux For You*, *Rails Magazine*, *Developer.com* (<http://www.developer.com/>), and *SitePoint Ruby* (<http://www.sitepoint.com/ruby/>). He currently resides in Taiwan. He wishes to continue writing and share his knowledge as much as possible with budding developers.

I would like to thank my parents, my sister, and my wife for being very understanding while I was writing this book. They have been pushing me to do better on this front and have inspired me to write more and more. I would also like to thank my boss for encouraging and supporting me during the process.

About the Reviewers

Gabriel Hilal is a full stack web developer who specializes in Ruby on Rails and related technologies. He has a bachelor's degree in Information Systems (Internet business) and a master's degree in Information Systems with Management Studies, both from Kingston University, London. During his time at the university, he developed a passion for Ruby on Rails and has since then done freelance work using behavior-driven development and agile methodologies to build high-quality Rails applications. Gabriel can be contacted on his website (www.gabrielhilal.com) or by e-mail at gabriel@gabrielhilal.com.

Andrew Montgomery-Hurrell is a software developer, hacker, and an all-round geek who enjoys everything from Dungeons and Dragons to DevOps. From an early age, he was fascinated with computers, and after cutting his teeth on BASIC with aging Amstrad CPCs and Amigas, he moved on to Linux admin, C/C++, followed by Python and then Ruby. Since the early 2000s, he has worked on a number of web applications in a range of languages and technologies, right from small company catalog sites to large web applications that serve thousands of people across the globe. Trained and interested in computing from the bottom up and coming from a background in electronics and computer interfacing, Andrew has experience in the full stack of computing technology, from ASICs to applications.

When he isn't working on web applications or infrastructure tools for gaming events and hosting company Multiplay, he can be found hacking code, reading or writing fiction, playing computer games, or slaying dragons with his wife, Laura.

Peter F. Philips is a software engineer, data scientist, and problem solver from New York City who now resides in San Francisco, CA. He is the founder of TechForProgress and cofounder of Planet (<http://planet.io/>) and Recognize (<https://recognizeapp.com/>) apps. Peter has been working with Ruby on Rails for seven years since Version 1.6. He is determined to use technology to improve the planet. In his spare time, Peter enjoys photography, hiking, rock climbing, and travelling to remote areas of the globe.

Philip De Smedt is a freelance full-stack developer and cofounder of Compete Hub, the definitive database of all endurance races. His main focus is on API-driven development using Rails and AngularJS. Philip is also the author of *Upgrading to Rails 4*, a step-by-step guide on upgrading your Rails 3 application to Rails 4. He is a Bitcoin and Dogecoin advocate and has spoken at multiple user groups on Rails and cryptocurrencies. When he's not coding or creating products, he likes to cycle, read books, or go for a run.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Project 1: A Social Recipe-sharing Website	9
Mission briefing	9
Creating mockups	11
Adding test data and creating tests	17
Adding categories	23
Creating and adding recipes	26
Creating associations – recipes and categories	29
Adding authentication	31
Beautifying your views	34
Mission accomplished	39
Hotshot challenges	39
Project 2: Conference and Event RSVP Management	41
Mission briefing	41
Creating and administrating events	44
Creating search-friendly URLs for events	49
Adding tags to events	52
Tagging-based search and tag cloud	57
Adding Gravatar for a user	61
Creating RSVPs for events	63
Adding event moderation	66
Creating "My events" to manage events created by users	72
Mission accomplished	75
Hotshot challenges	76

Table of Contents

Project 3: Creating an Online Social Pinboard	77
Mission briefing	77
Creating file uploads and image resizing	79
Creating an infinitely scrollable page	87
Creating a responsive grid layout	91
Adding a full-text search	95
Resharing the pins and creating modal boxes using jQuery	102
Enabling the application to send a mail	105
Securing an application from cross-site scripting or XSS	112
Mission accomplished	113
Hotshot challenges	113
Project 4: Creating a Restaurant Menu Builder	115
Mission briefing	115
Creating organizations with sign up	117
Creating restaurants, menus, and items	124
Creating user roles	130
Creating plans	134
Creating subdomains	139
Adding multitenancy and reusable methods	144
Creating a monthly payment model, adding a free trial plan, and generate a monthly bill	146
Exporting data to a CSV format	150
Mission accomplished	152
Hotshot challenges	152
Project 5: Building a Customizable Content Management System	153
Mission briefing	153
Creating a separate admin area	155
Creating a CMS with the ability to create different types of pages	160
Managing page parts	168
Creating a Haml- and Sass-based template	172
Generating the content and pages	177
Implementing asset caching	182
Mission accomplished	185
Hotshot challenges	186
Project 6: Creating an Analytics Dashboard using Rails and Mongoid	187
Mission briefing	187
Creating a MongoDB database	190
Creating a click-tracking mechanism	193

Table of Contents

Creating a visit-tracking mechanism	195
Writing map-reduce and aggregation to fetch and analyze data	199
Creating a dashboard to display clicks and impression values	205
Creating a line graph of the daily click activity	207
Creating a bar graph of the daily visit activity	210
Creating a demographic-based donut chart	213
Mission accomplished	218
Hotshot challenges	218
Project 7: Creating an API Mashup – Twitter and Google Maps	219
Mission briefing	219
Creating an application login with Twitter	221
Calling all Twitter friends	227
Getting latitude and longitude details of the user's location	232
Passing Twitter data to the Google Maps API using Rails	234
Displaying friends on the map using the Google API	237
Creating points of interest – filter users based on their location	241
Mission accomplished	247
Hotshot challenges	247
Project 8: API Only Application – Backend for a Mobile App	249
Mission briefing	249
Creating, editing, and deleting notes	251
Arranging notes category wise	261
Sending join data via JSON	264
Creating an OAuth2 provider	268
Generating API keys	273
Securing the application	279
Mission accomplished	282
Hotshot challenges	283
Project 9: Video Streaming Website using Rails and HTML5	285
Mission briefing	285
Uploading the video	287
Encoding the video	291
Displaying the video panel and playing the video	299
Caching the content – text and video	304
Queueing the job	310
Mission accomplished	316
Hotshot challenges	317

Table of Contents

Project 10: A Rails Engines-based E-Commerce Platform	319
Mission briefing	319
Creating a category and product listing	321
Creating a shopping cart and an Add to Cart feature	329
Packaging the engine as a gem	339
Mounting the engine on a blank Rails application	345
Customizing and overriding the default classes	349
Mission accomplished	354
Hotshot challenges	354
Index	355

Preface

In the past few years, Rails has emerged as one of the most popular choices of framework for developing web applications. It is also one of the most popular courses on all the major websites that teach web development, and a lot of developers have built a career out of it. Rails is known for providing productivity to developers and allows them to write clean and functional human-readable code. The latest major version of Rails, Rails 4, is a feature-packed update with a lot of new syntaxes and patterns.

Rails 4 Application Development Hotshot presents a practical approach to upgrade your Rails knowledge to Rails 4. This is done by building the most popular types of applications that people usually build using Rails and highlighting the new ways of doing this as opposed to the old ones in the latest version. The book also closely follows best practices and the commonly used gems and their compatibility with the latest Rails version. While working on these projects, we will also see some new design patterns and get ideas to refactor our current codebase. This book will help you write basic applications that are customizable and scalable and introduce you to a wide spectrum of concepts and ideas.

What this book covers

Project 1, A Social Recipe-sharing Website, explains how to create a website where many users can sign up, log in, create food recipes, and categorize them into different types.

Project 2, Conference and Event RSVP Management, explains how to create an application where users can create events, organize meetups for different topics and themes, and other users can join them in these events.

Project 3, Creating an Online Social Pinboard, covers how to create an online pinboard, where a user can pin whatever he/she likes on to it and organize these objects. These pins can be repinned by other users on to their pinboards and thus create an online collection of the things or objects that people like.

Project 4, Creating a Restaurant Menu Builder, covers how to build a fully responsive system to create and manage menus for a restaurant. This project will port restaurant menus to tablets and smartphones and also demonstrate how to make an SaaS application in Rails.

Project 5, Building a Customizable Content Management System, explains how to create a customizable content management system to power simple content-driven websites. We will effectively create a system where designers will have the freedom to choose the frontend they want and end users can easily manage the content for that frontend.

Project 6, Creating an Analytics Dashboard using Rails and Mongoid, will cover tracking clicks, page views, and the location of the visitors who read the content generated from the website. We will analyze the data and generate different types of graphs that represent different types of data.

Project 7, Creating an API Mashup – Twitter and Google Maps, will dive into an API mashup of Twitter and Google Maps that will generate an application to map the locations of your friends who are tweeting. We will also filter these people based on country names.

Project 8, API Only Application – Backend for a Mobile App, explains an application where the entire backend is in the form of an API. The entire data will be available on the frontend in the form of JSON through API endpoints. The frontend can be a web or mobile application.

Project 9, Video Streaming Website using Rails and HTML5, explains how to create an application to upload and encode videos. This application will allow visitors to stream and watch videos using an HTML5-based player.

Project 10, A Rails Engines-based E-Commerce Platform, explains how to create a Rails engine for generating an e-commerce application. This is mountable inside a blank Rails application.

What you need for this book

In order to work with the projects in this book, you will need the following installed on your system:

- ▶ Ruby 1.9.3
- ▶ Rails 4
- ▶ MySQL 5+
- ▶ MongoDB

- ▶ jQuery
- ▶ ImageMagick
- ▶ RMagick
- ▶ Git
- ▶ Morris.js
- ▶ Apache Solr
- ▶ Apache Tomcat
- ▶ Bootstrap
- ▶ Sass
- ▶ Sublime Text
- ▶ A tool for mock-ups
- ▶ Haml
- ▶ Memcached
- ▶ Twitter API keys
- ▶ Google Maps API keys
- ▶ The Rails API
- ▶ FFmpeg
- ▶ Redis
- ▶ Video.js
- ▶ A GitHub account
- ▶ Devise
- ▶ Doorkeeper

All projects have been upgraded and tested with Ruby 2.0 and Rails 4.1.0 beta.

Who this book is for

This book is aimed at developers who are already familiar with the basics of the Rails framework and have worked with Rails 3.2 or earlier versions. As the book follows a practical approach and uses terminology specific to Rails and web programming, it is assumed you have some prior experience with the development of applications. This book will help you upgrade your knowledge and improve its applicability.

Conventions

In this book, you will find several headings that appear frequently. To give clear instructions of how to complete a procedure or task, we use:

Mission briefing

This section explains what you will build, with a screenshot of the completed project.

Why is it awesome?

This section explains why the project is cool, unique, exciting, and interesting. It describes what advantage the project will give you.

Your Hotshot objectives

This section explains the eight major tasks required to complete your project:

- ▶ Task 1
- ▶ Task 2
- ▶ Task 3
- ▶ Task 4
- ▶ Task 5
- ▶ Task 6
- ▶ Task 7
- ▶ Task 8

Mission checklist

This section explains any prerequisites for the project, such as resources or libraries that need to be downloaded, and so on.

Task 1

This section explains the task that you will perform.

Prepare for lift off

This section explains any preliminary work that you may need to do before beginning work on the task.

Engage thrusters

This section lists the steps required in order to complete the task.

Objective complete - mini debriefing

This section explains how the steps performed in the previous section allow us to complete the task. This section is mandatory.

Classified intel

This section provides extra information that is relevant to the task.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In case the form validation fails, the `file` field is reset."

In all code blocks, the first line is the name of the file kept there for your reference, followed by the code. An example of a code block is shown as follows:

```
app/models/event.rb
class Event < ActiveRecord::Base
  belongs_to :organizers, class_name: "User"
end
```

Database migrations that appear in the book appear without the filename as the generated filename varies from system to system. Following is how it is defined in the book:

```
class AddPlanIdToUsers < ActiveRecord::Migration
  def change
    add_column :users, :plan_id, :integer
  end
end
```

Any command-line input or output is written as follows:

```
:~/pinpost$ rails g jquery:install
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "We are going to select **From Scratch** and build our wireframes using the given set of tools."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

Project 1

A Social Recipe-sharing Website

Food-recipe websites have been in existence since the advent of the Internet. Food.com, thefoodnetwork.com, and bbcgoodfood.com are some of most visited sites. Food is also one of the most popular searched categories on the Internet. Most of these websites have experts writing content for them. In this project, we will develop a website where amateur users can upload their recipes and those recipes can be viewed and shared by others and generated by several users. The recipes can be shared over various social networking sites by the readers.

Mission briefing

Our goal is to create a very basic social website where users can sign up and create recipes they know the best. Other users can filter these recipes based on their interests, tastes, and food preferences and share it on Facebook, Twitter, or other social networking sites of their choice. At the end of this project, we should be able to perform the following tasks:

- ▶ Create an application
- ▶ Know what's the best way for creating an application
- ▶ Make use of some of the new features available for creating the application

User stories are a very important part of the entire project. They can make or break project schedules and have a drastic effect on the product in the long run. Once defined, our use cases will have steps on how a user interacts with the application and the validations required for it to pass. It will be much easier for us to keep this as a reference while coding. A good specification, both visual and technical, goes a long way in helping developers save time.

The home page will contain feed of the entire system—users who have newly joined the system, created new recipes, and edited new recipes. The screenshot of the home page of the final system is as follows:

The screenshot shows a web application interface for a recipe sharing site named 'curry nation'. At the top right are 'Login' and 'SignUp' buttons. The main area has a header 'Recipes'. Below it is a table with columns: Title, Preference, Type, Cuisine, Servings, Time, Difficulty, Ingredients, and Procedure. Two rows of data are shown for 'Double Cheese Nachos':
Row 1: Double Cheese Nachos, Vegan, Mexican, 3, Hard, test, test, Show
Row 2: Double Cheese Nachos, Vegetarian, Italian, 3, Medium, test, test, Show
A 'Create a Recipe' button is located below the table. To the left of the table are three vertical sidebar menus:

- Food Preferences:** Vegetarian, Vegan, Meat, Dairy
- Food Type:** Curry, Dessert, Sides, Breakfast
- Cuisine:** Italian, Mexican, Indian, Chinese

Why is it awesome?

Everyone loves food, and some of us like to cook food too. The simplest and the most interesting way to build momentum for development is with a simple project. We will use this project to lay the foundation of Rails 4 comprehensively and build a base for the upcoming projects. Developers who have been using earlier versions of Rails will get a chance to work with new features in Version 4.0.0. Also, this will set the tone for the rest of the book, in terms of the process we will follow or we should follow while building our applications. We are following a test-driven development approach in the context of Rails 4. So, we will get a fair amount of exposure to the minitest framework, which has been newly introduced, and we will follow it up with some basics of ActiveRecord. While running through this, we will also work with Bootstrap 3.0 to style our views.

Your Hotshot objectives

While building this application, we will complete the following tasks:

- ▶ Creating mockups
- ▶ Adding test data and creating tests
- ▶ Adding categories

- ▶ Creating and adding recipes
- ▶ Creating associations – recipes and categories
- ▶ Adding authentication
- ▶ Beautifying your views

Mission checklist

We need the following software installed on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MySQL 6
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Devise
- ▶ Git
- ▶ A tool for mockups; I personally use MockFlow

Creating mockups

Before we actually start developing the application, we will build two types of specifications: visual specifications called mockups and technical specifications called user stories. Visual imagination needs a fair bit of creativity and is best left to the designers; however, for our reference here, we will see how to create mockups in case you are working on an end-to-end process.

Prepare for lift off

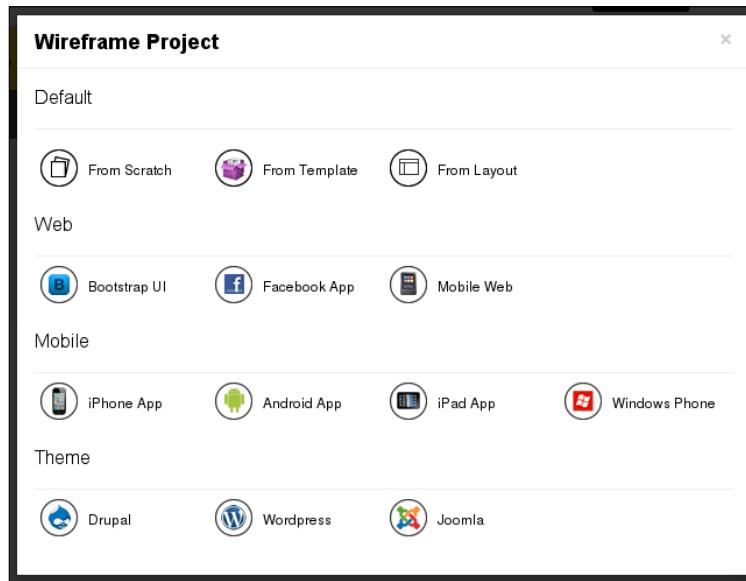
There are several mockup tools available online and are free to download and install.

Balsamiq (<https://www.mybalsamiq.com>), **MockFlow** (<http://mockflow.com>), and **mockingbird** (<https://gomockingbird.com/>) are some of the tools that I have explored and are fairly useful. We will use MockFlow for our projects. Sign up and create a free account with MockFlow.

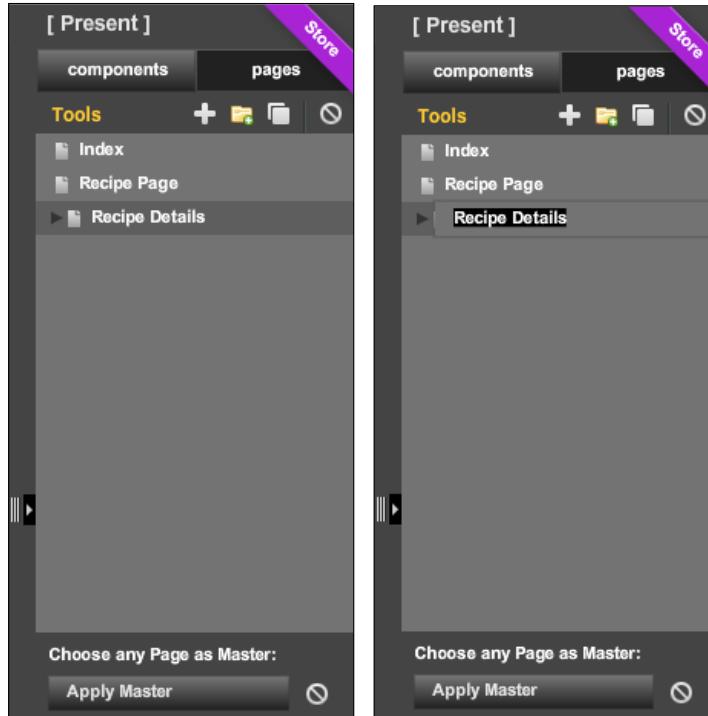
Engage thrusters

For creating mockups, we will perform the following steps:

1. Setting up a project in MockFlow is pretty straightforward. As soon as we log in to the account, we will be able to see an **Add Project** button. Once we click on it, the following screen shows up with various options for setting up different kinds of projects. We are going to select **From Scratch** and build our Wireframes using the given set of tools.

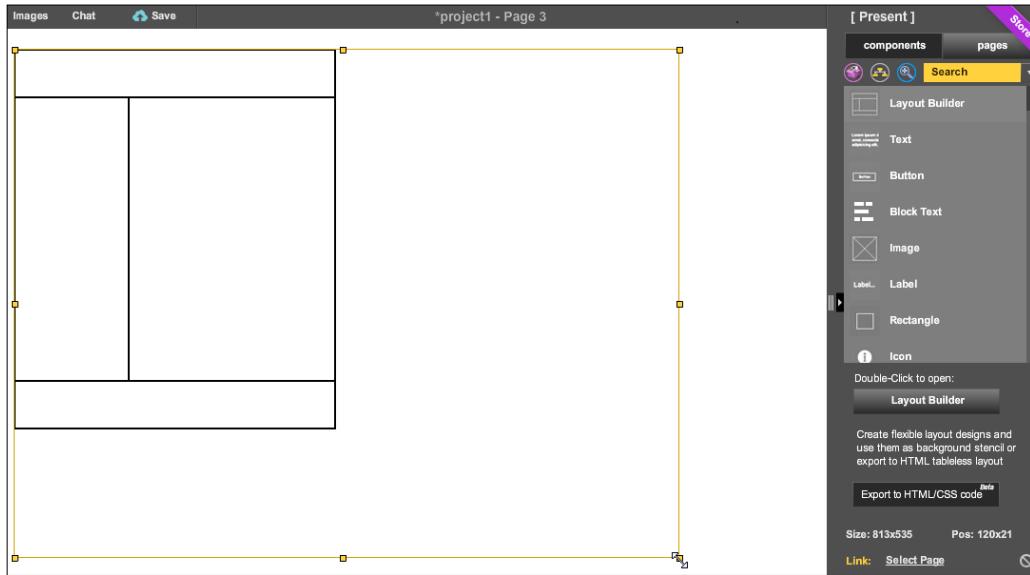


2. We will select the **From Scratch** option present under the **Wireframe Project** screen, name it, and proceed with the setup of the pages we want in our application.
3. The tool to the right contains two tabs:
 - pages**: With this option, you can **Create**, **Sort**, **Duplicate**, and **Delete** pages in your application
 - components**: With this option, the textboxes, text areas, scrollbars, logos, images, and different elements of the page can be simply dragged-and-dropped from the **component** panel to the canvas on the center of the page to create a Wireframe

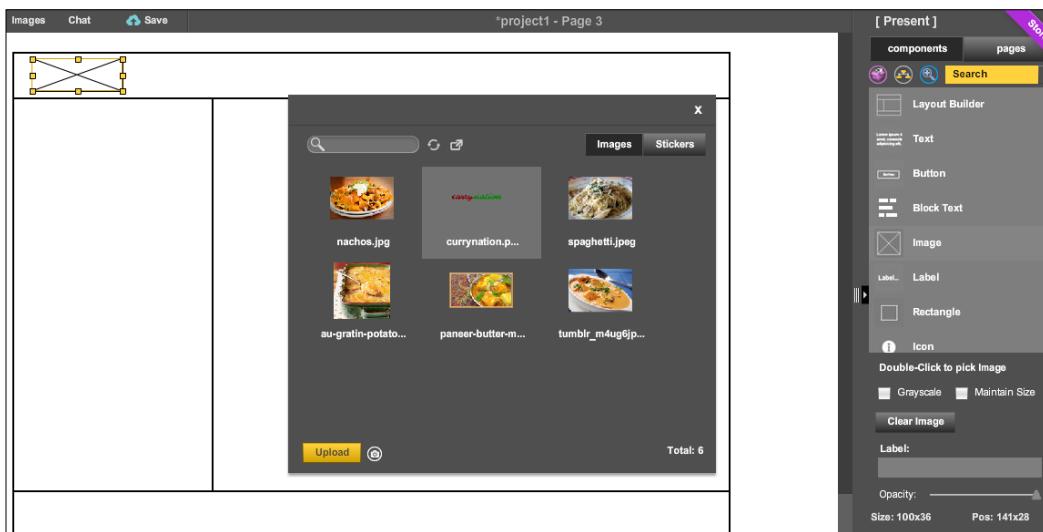


4. Let's start building our first mockup. Drag-and-drop the **Layout Builder** icon located in the **components** panel, and using your mouse, create and resize it so it fits on the page.
5. This layout suits our application needs because our aim is to build an application with a filter bar to the left that would allow users to filter categories with ease. The central portion will display the content and will contain the list of various recipes. The portion to the left will contain the list of various categories.

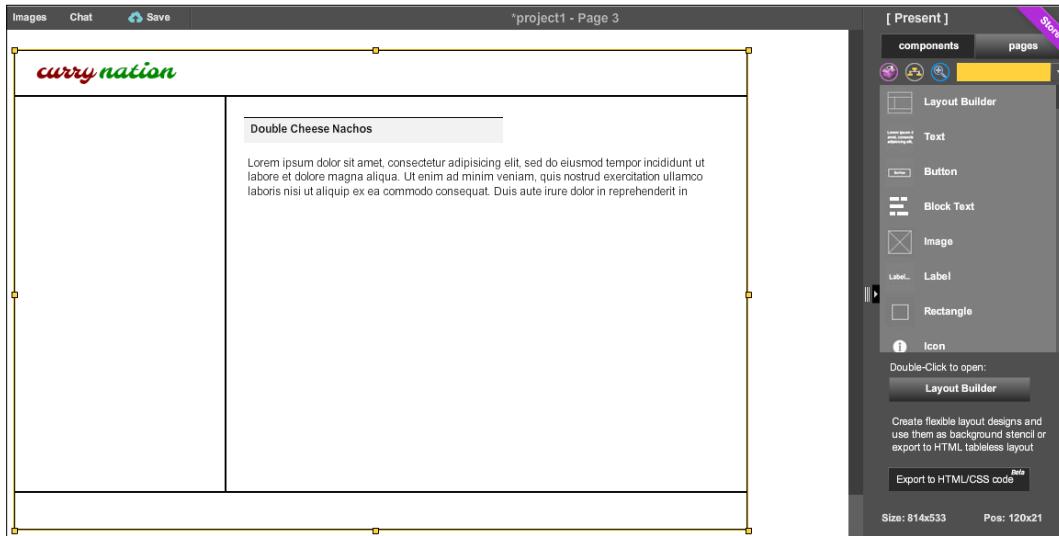
6. The header will contain the logo, login details, and dashboard links, whereas the footer will contain copyright information and company information links.



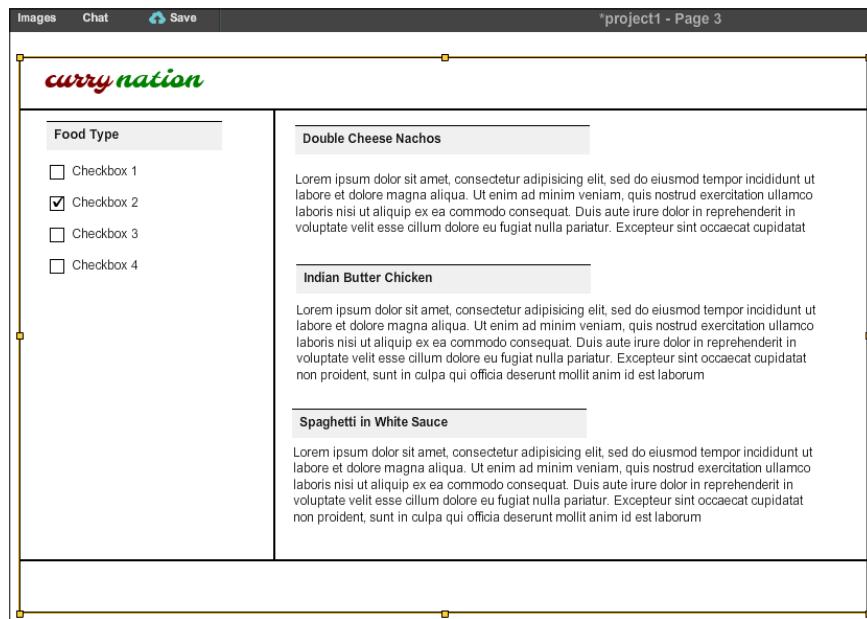
7. After resizing the layout builder, we will add the logo and images to the header. In order to do so, we will first drag-and-drop the **Image** component from the **components** panel and double-click on it. We will be presented with a modal box to manage and upload images. Browse and upload images using this tool. Once an image is selected, just drag and move it to the position where you want to see the logo placed.



8. The next step would logically be to build the inner page. This page will have some text on it. We will drag the title and text from the **components** bar and drop it to the central part of the layout.



9. Add checkboxes and the remaining elements to the mockup.



10. We will finally add some checkboxes to the left bar for filters. This includes food type, food preferences, and cuisines in order to properly categorize our recipes.
11. We can now figure out other elements of the page, for example, in order to create links such as **Login/Signup**, and **About Us**, we can use the **Label** component from the **components** panel.

Objective complete – mini debriefing

As seen in the previous steps, we added various page elements, including text areas, a title, and checkboxes to our page. We can use these page elements to create mockups for all the pages. Mockups for the home page and recipe page are shown in the following two screenshots:

The screenshot shows a web application interface for 'curry nation'. At the top, there's a navigation bar with 'Login / Signup' on the right. Below it is a sidebar containing three sections: 'Food Type' (Curry, Dessert checked, Sides, Breakfast), 'Food Preference' (Vegetarian, Vegan checked, Meat, Dairy), and 'Cuisine' (Italian, Mexican checked, Indian, Chinese). To the right of the sidebar, there are three recipe cards: 'Double Cheese Nachos' (Lorem ipsum placeholder text), 'Indian Butter Chicken' (Lorem ipsum placeholder text), and 'Spaghetti in White Sauce' (Lorem ipsum placeholder text). At the bottom of the page, there are links for 'About Us', 'Company', and 'Contact Us' on the left, and 'Copyright 2013 Currynation LLC' on the right.

The home page now looks complete with different links and information in the footer shown as follows:

The screenshot shows a website layout for 'curry nation'. At the top, there's a navigation bar with links for 'John Doe', 'Profile', and 'Logout'. On the left, there's a sidebar with 'Similar Recipes' and two items: 'Paneer Butter' and 'Kadhai Chicken'. The main content area is titled 'Indian Butter Chicken'. It includes sections for 'Ingredients' (listing Boneless Chicken, Butter, Salt, 2 Tomatoes, and Olive Oil) and 'Procedure' (with two steps). Each step contains placeholder text ('Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud')). At the bottom of the page, there are links for 'About Us', 'Company', and 'Contact Us', along with a copyright notice: 'Copyright 2013 CurryNation LLC'.

Classified intel

The options offered in MockFlow include building mockups for the following:

- ▶ Web applications
- ▶ Mobile applications
- ▶ Themes specific to a particular CMS, or using a particular CSS framework such as Bootstrap
- ▶ Simple Wireframing from scratch or from templates

Adding test data and creating tests

Rails does a lot of work for us by providing us with generators, right from a blank application to different parts of the application. The trick lies in using it only when required. Our first application will consider a very simple use case of generators, but we will scarcely use them in subsequent projects. In this task, we will generate our application and write tests before we write the code.

Prepare for lift off

As MySQL and PostgreSQL are the most common RDBMS around, we're going to use either of them for building most of our applications. The default database in the development mode with Rails is SQLite. Make sure you have one of these databases working on your system and also make sure that the connection with Rails is working. We will use MySQL for most of our projects including this one.

Engage thrusters

The steps for creating a new application and setting up the **database (db)** are as follows:

1. Let us first create a blank application with a MySQL database as the default database using the following command:

```
:~/\$ rails new curry-nation -d mysql
```

2. Now we can go ahead and set up the application's `database.yml` file under `config` to connect to the system's database. You would need to make this file suit the database that you are using. We are using MySQL; likewise, you can edit the file for the database of your choice.

```
config/database.yml
development
  adapter: mysql2
  encoding: utf8
  database: curry-nation_development
  pool: 5
  username: root
  password:
  socket: /var/run/mysqld/mysqld.sock
test:
  adapter: mysql2
  encoding: utf8
  database: curry-nation_test
  pool: 5
  username: root
  password:
  socket: /var/run/mysqld/mysqld.sock

production:
  adapter: mysql2
  encoding: utf8
  database: curry-nation_production
  pool: 5
  username: root
  password:
  socket: /var/run/mysqld/mysqld.sock
```

3. Once the database is set up, we need to create the database using the following commands:

```
:~/curry-nation$ rake db:create  
:~/curry-nation$ rake db:migrate
```

4. We will first prepare our fixtures. Fixtures contain test data that loads into the test database. These are placed in the fixtures folder under test with the filename recipes.yml:

```
test/fixtures/recipes.yml  
curry:  
  title: Curry  
  food_preference_id: 1  
  food_type: 1  
  cuisine_id: 1  
  servings: 1  
  cooking_time: 1  
  level_of_difficulty: Easy  
  ingredients: Onions Tomatoes Salt Oil  
  procedure: Heat Oil Chop Onions, tomatoes and  
    Salt to it.
```

5. Once the fixtures are ready, we can populate the db with fixtures. However, we have not yet created the models and tables. Hence, we will load the fixtures' data once we create our models.

6. We can now go ahead and write integration tests. We will now add an integration test and create it line by line:

```
~/test/integration$ recipe_test.rb
```

7. We will load the test helper that will load the test database and other dependencies for the test:

```
require 'test_helper'
```

8. Load the test record and navigate to the new recipe page:

```
test/integration/recipe_test.rb  
  curry = recipes(:curry)  
  get "/recipes/new"
```

9. Post the data to the new method and assert for a success response. At this point, it even checks for validations if they are defined. Depending on this, it would be redirected to the index page:

```
test/integration/recipe_test.rb  
  assert_response :success  
  post_via_redirect "/recipes/new", title:  
    recipes(:curry).title
```

10. We can now prepare the database and run the test:

```
:~/curry-nation/test/integration$ rake db:create RAILS_ENV="test"  
(in /curry-nation)  
r:~/curry-nation/test/integration$ rake test recipe_test.rb  
 10 tests, 10 assertions, 10 failures, 0 success, 0 skips
```

11. The final integration test looks like this:

```
test/integration/recipe_test.rb  
class RecipeFlowsTest < ActionDispatch::IntegrationTest  
  fixtures :recipes  
  test "create recipes" do  
    https!  
    curry = recipes(:curry)  
    get "/recipes/new"  
    assert_response :success  
    post_via_redirect "/recipes/new", title:  
      recipes(:curry).title  
    assert_equal '/recipes', path  
    assert_equal 'Create Recipe', flash[:notice]  
    https!(false)  
    get "/recipes"  
    assert_response :success  
    assert assigns(:recipes)  
  end  
end
```

12. Our integration tests look at the way the pages and routes work with each other. Controller tests look at how data is passed between these calls, and the methods call themselves.

13. Set up a recipe variable and get the index method:

```
test/integration/recipe_test.rb  
class RecipesControllerTest < ActionController::TestCase  
  setup do  
    @recipe = recipes(:one)  
  end  
  test "should get index" do  
    get :index  
    assert_response :success  
    assert_not_nil assigns(:recipes)  
  end
```

14. We will use assert to get a new page in our controller test:

```
test/controllers/recipes_controller_test.rb  
  test "should get new" do  
    get :new  
    assert_response :success  
  end
```

15. We will also perform a test for creating a recipe:

```
test/controllers/recipes_controller_test.rb
  test "should create recipe" do
    assert_difference('Recipe.count') do
      post :create, recipe: { cooking_time:
        @recipe.cooking_time, cuisine_id:
        @recipe.cuisine_id, food_preference_id:
        @recipe.food_preference_id, food_type:
        @recipe.food_type, ingredients:
        @recipe.ingredients, level_of_difficulty:
        @recipe.level_of_difficulty, procedure:
        @recipe.procedure, servings: @recipe.servings,
        title: @recipe.title }
    end

    assert_redirected_to recipe_path(assigns(:recipe))
  end
```

16. Add a test for showing a recipe using the following code:

```
test/controllers/recipes_controller_test.rb
  test "should show recipe" do
    get :show, id: @recipe
    assert_response :success
  end
```

17. We will test the edit and update methods:

```
test/controllers/recipes_controller_test.rb
  test "should get edit" do
    get :edit, id: @recipe
    assert_response :success
  end

  test "should update recipe" do
    patch :update, id: @recipe, recipe: { cooking_time:
      @recipe.cooking_time, cuisine_id:
      @recipe.cuisine_id, food_preference_id:
      @recipe.food_preference_id, food_type:
      @recipe.food_type, ingredients:
      @recipe.ingredients, level_of_difficulty:
      @recipe.level_of_difficulty, procedure:
      @recipe.procedure, servings: @recipe.servings,
      title: @recipe.title }
    assert_redirected_to recipe_path(assigns(:recipe))
  end
```

18. Lastly, we will check for deletions:

```
test/controllers/recipes_controller_test.rb
  test "should destroy recipe" do
    assert_difference('Recipe.count', -1) do
      delete :destroy, id: @recipe
    end

    assert_redirected_to recipes_path
  end
end
```

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Objective complete – mini debriefing

As we saw in the previous task, the structure of the default testing framework in Rails 4 includes the respective style folder structure, which is much cleaner and nicely abstracted compared to the earlier versions. This is how it looks:

```
:~/curry-nation/test$ ls
controllers fixtures helpers integration mailers models
test_helper.rb
```

The test folder is self-descriptive in terms of the folder structure and clearly denotes which test belongs to which part of the system.

Here, we have prepared the test data and written tests that match the specifications. This will help us emulate our functionality. We are now ready to write some code in order to run our tests. The tests in this case failed because there is no code for the tests to run.

Classified intel

Testing is the backbone of your application. If you don't write tests, you are opening a Pandora's box for yourself.

Adding categories

To make the content of the website easily browsable, it makes sense to categorize it in different ways according to the diversity of choice a user has regarding food recipes. In this task, we will build navigation bars that would be visible on the left-hand side. Actually, it goes much deeper than just being the navigation bar. This is because it has to be built in a way that allows us to effectively search for data in future. So, for us, categories are a way to arrange data and make it more accessible, and in this task, we will see how to create categories.

Categories in our application are divided into three parts:

- ▶ **Food preferences:** Food preferences include the value system of users. They might like dairy free, vegan, vegetarian, meat, and so on. Recipes are categorized on the basis of this.
- ▶ **Food types:** Food types denote whether the food is a main course, a curry, a side dish, or a dessert.
- ▶ **Cuisines:** The final categorization is on the basis of cuisine.

Engage thrusters

The steps for adding categories are as follows:

1. We first need to create models that can be associated with the recipes:

```
:~/curry-nation$ rails g model food_type name:string
      invoke  active_record
      create   db/migrate/20130803103254
      _create_food_types.rb
      create     app/models/food_type.rb
      invoke    test_unit
      create     test/models/food_type_test.rb
      create     test/fixtures/food_types.yml
```

2. We can't leave the categories blank, and they need some default data. We do not have an interface to load categories so we will use the seeds' data by adding default data using seed scripts.
3. This generates a food type model, fixtures, blank tests, and table migrations. These values have to be available in the database in order to be used with the recipes. We will load them using seeds.rb.

```
db/seeds.rb
food_types = ["Curry", "Dessert", "Sides", "Breakfast"]
food_types.each{|d| FoodType.where(:name => d).create}
```

Once done, we'll run the following code:

```
rake db:migrate  
rake db:seed
```

The following steps will help us to modify seeds:

1. The default seeds, if simply defined, can create duplicate records in the database and might fail validations. This is because every time we run `rake db:seeds`, it runs all the queries again. In order to avoid this, we can add `first_or_create` after the data, which checks for the record in the database before adding it to the database:

```
db/seeds.rb  
food_types.each{|d| FoodType.where(:name => d).first_or_create}
```

2. Likewise, we can create other models related to categories in the same way:

```
:~/curry-nation$ rails g model food_preference  
  name:string  
    invoke active_record  
    create  
      db/migrate/20130803110704_create  
      _food_preferences.rb  
    create app/models/food_preference.rb  
    invoke test_unit  
    create test/models/food_preference_test.rb  
    create test/fixtures/food_preferences.yml  
:~/curry-nation$ rake db:migrate  
== CreateFoodPreferences: migrating =====  
=====  
-- create_table(:food_preferences)  
 -> 0.1313s  
== CreateFoodPreferences: migrated (0.1315s) =====  
=====  
  
:~/curry-nation$ rails g model cuisine name:string  
  invoke active_record  
  create  
    db/migrate/20130803111845_create_cuisines.rb  
  create app/models/cuisine.rb  
  invoke test_unit  
  create test/models/cuisine_test.rb  
  create test/fixtures/cuisines.yml  
:~/curry-nation$ rake db:migrate  
== CreateCuisines: migrating =====  
=====  
-- create_table(:cuisines)  
 -> 0.1107s  
== CreateCuisines: migrated (0.1109s) =====  
=====
```

3. Load them into the database as follows:

```
db/seeds.rb
food_preferences = ["Vegetarian", "Vegan",
    "Meat", "Dairy"]
food_preferences.each{|d| FoodPreference.where(:name =>
    d).first_or_create}

cuisines = ["Italian", "Mexican", "Indian", "Chinese"]
cuisines.each{|d| Cuisine.where(:name =>
    d).first_or_create}
:~/curry-nation$ rake db:seed
```

4. For accessing the console and checking the entered data, we can load the Rails console and check whether all the values are present in the database or not:

```
:~/curry-nation$ rails c
Loading development environment (Rails 4.0.0)
1.9.3-p327 :002 > FoodType.all
FoodType Load (0.9ms)  SELECT `food_types`.* FROM
`food_types`
=> #<ActiveRecord::Relation [#<FoodType id: 1, name:
    "Curry", created_at: "2013-08-03 10:57:37",
    updated_at: "2013-08-03 10:57:37">, #<FoodType id: 2,
    name: "Dessert", created_at: "2013-08-03 10:57:37",
    updated_at: "2013-08-03 10:57:37">, #<FoodType id: 3,
    name: "Sides", created_at: "2013-08-03 10:57:37",
    updated_at: "2013-08-03 10:57:37">, #<FoodType id: 4,
    name: "Breakfast", created_at: "2013-08-03 10:57:37",
    updated_at: "2013-08-03 10:57:37">] >
1.9.3-p327 :003 > FoodPreference.all
FoodPreference Load (0.7ms)  SELECT
`food_preferences`.* FROM `food_preferences`
=> #<ActiveRecord::Relation [#<FoodPreference id: 1,
    name: "Vegetarian", created_at: "2013-08-03
    11:15:56", updated_at: "2013-08-03 11:15:56">,
    #<FoodPreference id: 2, name: "Vegan", created_at:
    "2013-08-03 11:15:56", updated_at: "2013-08-03
    11:15:56">, #<FoodPreference id: 3, name: "Meat",
    created_at: "2013-08-03 11:15:56", updated_at: "2013-
    08-03 11:15:56">, #<FoodPreference id: 4, name:
    "Dairy", created_at: "2013-08-03 11:15:56",
    updated_at: "2013-08-03 11:15:56">] >
1.9.3-p327 :004 > Cuisine.all
Cuisine Load (0.6ms)  SELECT `cuisines`.* FROM
`cuisines`
```

```
=> #<ActiveRecord::Relation [#<Cuisine id: 1, name: "Italian", created_at: "2013-08-03 11:28:54", updated_at: "2013-08-03 11:28:54">, #<Cuisine id: 2, name: "Mexican", created_at: "2013-08-03 11:28:54", updated_at: "2013-08-03 11:28:54">, #<Cuisine id: 3, name: "Indian", created_at: "2013-08-03 11:28:54", updated_at: "2013-08-03 11:28:54">, #<Cuisine id: 4, name: "Chinese", created_at: "2013-08-03 11:28:54", updated_at: "2013-08-03 11:28:54">] >
```

Objective complete – mini debriefing

We have successfully created category-related models and loaded values to them using seeds. We also saw the best practice for creating seeds so that we can avoid loading duplicate data in the database.

Seeds should be defined for all kinds of default data in the system. Also, the process of adding seeds should be incremental and ongoing. Some might argue that it is very close to fixtures; however, fixtures belong to the test bed, whereas seeds are generic data that should be loaded by default in the system.

Creating and adding recipes

Scaffolding is the easiest way to start, but as the word itself suggests, it's just scaffolding. Rails goes much beyond that. Also, whether to use it or not in practical projects is a huge debate. However, I feel that we can use it to start but it's important that we build our functionalities in it. This will provide us with a template that adheres to best practices to start with, and then builds our code upon it.

Engage thrusters

After successfully writing our tests, we will write our code to make sure our tests run.

1. We will first understand our use case:

- User story; that is, to create a recipe
- User enters the title
- Users selects food preferences, food type, cuisine, and the level of difficulty
- User enters servings, cooking time, ingredients, and procedure
- User saves the recipe



Make sure that the validations required are as follows:

- ▶ Title is mandatory
- ▶ Food preferences, food type, and cuisine are mandatory

2. We will start with generating a scaffold. The general format is to write the command followed by the name of model, fields, and datatype of each field shown as follows:

```
:~/curry-nation$ rails g scaffold recipe title:string  
cooking_time:string difficulty_level:string  
food_type_id:integer food_preference_id:integer  
cuisine_id:integer ingredients:text procedure:text
```

This will create files that include model, controller, basic views, and skeleton tests.

3. We can now see what we have already created. In order to see what we have created so far, let's fire up our server and see what we just created by navigating to localhost:3000.

The screenshot shows a web browser window with the URL `localhost:3000/recipes/new` in the address bar. The page title is "New recipe". Below the title, there are eight input fields with labels: "Title", "Type", "Cuisines", "Cooking time", "Difficulty level", "Category", "Ingredients", and "Procedure". At the bottom of the form is a "Create Recipe" button.

4. Now, as we can see, the category values that we added previously are blank textboxes in our form. We would need to create dropdowns for each one of them so that they are selected and sent to the db.

```
<div class="form-group">
<%= f.label :food_preference %><br>
<%= f.select(:food_preference_id,
  options_from_collection_for_select(FoodPreference.all,
  :id, :name), {:prompt => 'Please Choose'}, :class =>
  "form-control") %>
</div>

<div class="form-group">
<%= f.label :food_type %><br>
<%= f.select(:food_type_id,
  options_from_collection_for_select(FoodType.all, :id,
  :name), {:prompt => 'Please Choose'}, :class => "form-
control") %>
</div>

<div class="form-group">
<%= f.label :cuisine %><br>
<%= f.select(:cuisine_id,
  options_from_collection_for_select(Cuisine.all, :id,
  :name), {:prompt => 'Please Choose'}, :class => "form-
control") %>
</div>
```

5. As you can see in the preceding code, we are able to populate the values in the select box from our database tables and pass the IDs to the recipe table of the database.
6. We will define an array in the recipe model and access it in the view. There is also another dropdown required for "level of difficulty" to be defined inside the recipe model. We can create a simple array with the names of difficulty levels as follows:

```
app/models/recipe.rb
DIFFICULTY=%w(Easy Medium Hard)
```

7. We can now call the level of difficulties directly inside our views and access the array values by calling it on the model name using `Recipe::DIFFICULTY`, shown as follows:

```
app/views/recipes/_form.html.erb
<div class="form-group">
<%= f.label :difficulty_level %><br>
<%= f.select :difficulty_level, Recipe::DIFFICULTY, {} ,
:label => "form-control" %>
</div>
```

New recipe

Title

Food preference

Food type

Cuisine

Servings

Cooking time

Level of difficulty

Ingredients

Objective complete – mini debriefing

At the end of this task, we will be able to create a recipe and add them to categories. However, we have not yet created a real association between them as we discussed earlier. We also saw that we can define arrays and call them directly from our `model` class like we did in the case of food type, food preferences, and cuisines.

Creating associations – recipes and categories

Associations are important in order to pass and access data between the models. ActiveRecord adds one of the major productivity boosts by avoiding writing SQL by hand. In this task, we will define relationships between different models and tell them how they should behave with each other.

Engage thrusters

We will discuss creating an association between the recipe and category models in this section.

1. According to our use case, each food type can have multiple recipes associated to it. This is because logically speaking, a category will have many recipes associated to it.

```
app/models/food_type.rb
class FoodType < ActiveRecord::Base
  has_many :recipes
end
```

2. Also, each recipe belongs to a particular food type, which we can define by adding a belongs_to rule to the recipe model.

```
app/models/recipe.rb
class Recipe < ActiveRecord::Base
  belongs_to :food_type
end
```

3. In the same way, we can associate other categories to the recipe model too, shown as follows:

```
app/models/food_preference.rb
class FoodPreference < ActiveRecord::Base
  has_many :recipes
end
app/models/cuisine.rb
class Cuisine < ActiveRecord::Base
  has_many :recipes
end
```

4. We can now display these values in our views in the following ways:

```
app/views/recipes/index.html.erb
<td><%= recipe.food_preference.name %></td>
<td><%= recipe.food_type.name %></td>
<td><%= recipe.cuisine.name %></td>
```

Objective complete – mini debriefing

We have successfully set up associations between the models, and they can now be accessed seamlessly between controllers and views.

Right associations are not only important for properly passing data between controllers and models, but also for critical tasks such as searching.

Adding authentication

We want legitimate people to post on our website and avoid spam. In order to do so, authentication is a must. In this task, we will see how to use **devise** to add authentication to the application. The choice of devise is quite obvious because it is a very complete authentication engine in every sense. It is also very easily extensible and hence the best choice for this.

Prepare for lift off

Devise is the most popular and up-to-date solution of authentication with Rails. We will use it to add user authentication to our website.

Engage thrusters

Let's have a look at how can we use devise to add user authentication to our website.

1. The use case for devise is as follows:

- ❑ User story; that is, user sign-up
- ❑ User clicks on sign-up
- ❑ User fills in the e-mail
- ❑ User enters and confirms the password
- ❑ If validations are passed, the user gets a valid account

The points that are checked for validations are:

- ▶ Is the e-mail format valid?
- ▶ Does the password comprise a minimum of eight characters in length?
- ▶ Does the information entered in the password and confirm password fields match?

2. We can add devise and generate the basic authentication by adding the following code to the `Gemfile` and running the bundle:

```
gem 'devise', github: 'plataformatec/devise'
```

3. We can install devise using the following command line. We can then go ahead and perform the installation of basic configuration files of devise:

```
:~/curry-nation$rails g devise:install
```

4. This will create two files for us: `initializers/devise.rb` and `locales/devise.en.yml`. We can now generate our user model:

```
:/curry-nation$rails g devise user
```

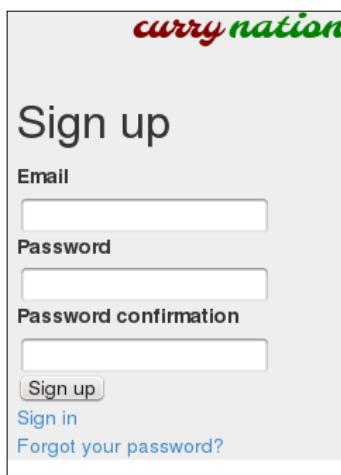
5. The following command line will mount the Devise application routes on the `routes.rb` file:

```
config/routes.rb  
  devise_for :users
```

6. We will now protect selected methods. Devise comes with a set of methods that can be readily used with user-related resources in our application. We will first proceed with the protection of our specific methods inside our recipe model:

```
app/controllers/recipes_controller.rb  
  before_filter :authenticate_user!, only: [:new, :edit, :create,  
  :update, :destroy]
```

7. This will allow us to protect the `new`, `edit`, `create`, `update`, and `destroy` methods using user authentication. The `current_user` method allows access to the logged-in user in the session. We can display the e-mail of the user using this method.



8. Let's write a "create user login" user story as follows:

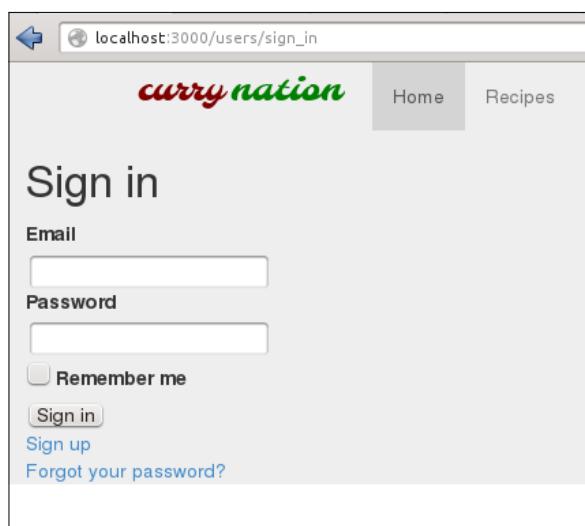
- ❑ User story; that is, user login
- ❑ User clicks on the login link
- ❑ User fills in the username and password
- ❑ Validations are applied to check whether both the username and password are present in the database

9. We can also protect specific methods in views. The `if user_signed_in?` method is a conditional method provided by Devise. We can use it to check whether the user session is in progress or not. If it is, then we can display the e-mail of the user and the logout link; if not, then display the login and sign-up links:

```
app/views/layouts/application.html.erb
<ul class="nav navbar-nav pull-right">
  <% if user_signed_in? %>
    <li><%= link_to "#{current_user.email}" %></li>
    <li class="active"><%= link_to "Logout",
      destroy_user_session_path%></li>
  <%else%>
    <li><%= link_to "Login", new_user_session_path %></li>
    <li class="active"><%= link_to "SignUp",
      new_user_registration_path%></li>
  <%end%>
</ul>
```

10. We can make the methods visible only to the logged-in users. Also, though we have already protected our `new` and `edit` methods using authentication, we can hide them altogether from the views, again by using the `if user_signed_in?` method:

```
app/views/recipes/index.html.erb
<% if user_signed_in? %>
  <td><%= link_to 'Edit', edit_recipe_path(recipe),
    :class=>"btn btn-success btn-small" %></td>
  <td><%= link_to 'Delete', recipe, method: :delete,
    data: { confirm: 'Are you sure?' }, :class=>
    "btn btn-danger btn-small" %></td>
<%end%>
```



Objective complete – mini debriefing

At the end of this task, our application has devise-based authentication for login and sign-up functionalities. We also protected certain methods and made them accessible only after we completed the login process. Lastly, we looked at various methods to pass user data to session objects such as `current_user`.

Devise also supports OpenLDAP and API methods for extending authentication for our apps on the mobile platform.

Beautifying your views

Proper styling is equally important as it can make or break your website despite writing all of the code correctly. With a myriad of websites at a user's disposal and so many new user experiences, the user interface takes on a huge role.

We will use Twitter's Bootstrap framework not only for our convenience, but also to ensure good quality code for the markup. The main advantages Bootstrap has to offer are as follows:

- ▶ Clean and high performing markup
- ▶ Responsiveness
- ▶ HTML4 and HTML5 doctype standards compliant
- ▶ Easily customizable
- ▶ Uses the latest design practices

Prepare for lift off

Read Bootstrap's *Getting started* task at <http://getbootstrap.com/getting-started/> and get started with Version 3.

Engage thrusters

In this task, we will see some of the styling classes of Bootstrap and use it to style our application:

1. Add Bootstrap to the asset pipeline.

We will use the `bootstrap-rails` gem in order to add Bootstrap to our asset pipeline. Add the following line to the `Gemfile` and `bundle install`:

```
gem 'anjlab-bootstrap-rails', :require => 'bootstrap-rails'
```

2. Make CSS and JavaScript available to the asset pipeline. Then add the following line to `application.css`. This is for informing the asset pipeline to access Bootstrap files from this folder:

```
app/assets/application.css
 *= require twitter/bootstrap
```

3. Add the required directive to the `application.js` file to make all the Bootstrap JavaScripts available to the Rails application:

```
app/assets/application.js
 // = require twitter/bootstrap
```

Then add the necessary style to the layouts.

4. Once we've added these, all the CSS and `.js` files in Bootstrap are ready to be used in our application. This is how our code looks at the moment. There is practically no styling and only the default methods of `scaffold.css` are being used:

```
app/views/layouts/application.html.erb
<body>
  <ul>
    <li><%= link_to "Recipes", recipes_path %></li>
  </ul>
  <ul>
    <% if user_signed_in? %>
    <li><%= link_to "#{current_user.email}" %></li>
    <li><%= link_to "Logout",
      destroy_user_session_path%></li>
    <%else%>
    <li><%= link_to "Login", new_user_session_path
      %></li>
    <li><%= link_to "SignUp",
      new_user_registration_path%></li>
    <%end%>
  </ul>
  <%= yield %>
</body>
```

5. Create a layout that consists of two columns.
6. According to our mockup, we intend to make a two-column layout for our application. The left bar contains various categories, and the central portion is present for rendering the content.
7. Bootstrap does this by creating rows and then dividing them into columns of different sizes. All these classes are inherited from a class called `container`, which has all body-related classes:

```
app/views/layouts/application.html.erb
<div class="container">
  <div class="row">
```

```
<div class="col-lg-2">
</div>
<div class="col-lg-9">
</div>
</div>
</div>
```

8. The `col-lg-2` class will create a `div` tag with a width of 16.6667 percent, and `col-lg-9` will create a `div` tag with a width of 75 percent.
9. Then you can style the navigation. The top-level class for creating a navigation bar is `navbar`, and the specific class to create a menu that sticks to the top is `navbar-static-top`. The `navbar-brand` class is the logo class:

```
app/views/layouts/application.html.erb
<!-- Static navbar -->
<div class="navbar navbar-static-top">
<div class="container">
<button type="button" class="navbar-toggle"
 data-toggle="collapse" data-target=".nav-collapse">
<span class="icon-bar"></span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>
</button>
<a class="navbar-brand" href="#"><%= image_tag
 "currynation.png"%></a>
<div class="nav-collapse collapse">
<ul class="nav navbar-nav">
<li class="active"><%= link_to "Recipes",
 recipes_path %></li>
</ul>
<ul class="nav navbar-nav pull-right">
<% if user_signed_in? %>
<li><%= link_to "#{current_user.email}" %></li>
<li class="active"><%= link_to "Logout",
 destroy_user_session_path%></li>
<% else %>
<li><%= link_to "Login", new_user_session_path
 %></li>
<li class="active"><%= link_to "SignUp",
 new_user_registration_path%></li>
<% end %>
</ul>
</div><!-- .nav-collapse -->
</div>
```

10. Add styles to individual pages and the customizing buttons.
11. We will also style our index page by inheriting the `table` class:

```
<table class="table">
```

12. We can also customize the links in our app so they look like buttons by adding a class called `btn`, following the `btn-primary` class, which defines the color and size of the button. So, for example, we will apply the color blue to the button and assign a small size to it using the `btn-small` class:

```
app/views/recipes/index.html.erb
<td><%= link_to 'Show', recipe, :class=>"btn btn-
primary btn-small"%></td>
<% if user_signed_in? %>
<td><%= link_to 'Edit', edit_recipe_path(recipe),
:class=>"btn btn-success btn-small"%></td>
<td><%= link_to 'Delete', recipe, method: :delete,
data: { confirm: 'Are you sure?' }, :class=>"btn
btn-danger btn-small" %></td>
<%end%>
```

13. Now we'll look at how to style sublinks and wrap them into Rails' loops:

Our left bar for displaying categories can be displayed as a panel with several sublinks. These are generated in loop using the Rails' `each` loop. We will first define the values for the sidebar in our `application_controller.rb` file:

```
app/controllers/application.rb
helper_method :sidebar_values

def sidebar_values
@food_preferences = FoodPreference.all
@food_types = FoodType.all
@cuisines = Cuisine.all
end
```

In `app/views/layouts/application.html.erb`, we must define the sidebar as rows :

```
app/views/layouts/application.html.erb
<div class="row">
<div class="col-lg-2">
<div class="panel panel-primary" id="panels">
<div class="panel-heading">Food Preferences
</div>
<% @food_preferences.each do |fp| %>
<p><%= fp.name%></p>
<%end%>
</div>
```

A Social Recipe-sharing Website

```
<div class="panel panel-primary" id="panels">
  <div class="panel-heading">Food Type
  </div>
  <% @food_types.each do |ft| %>
<p><%= ft.name%></p>
<%end%>
</div>
  <div class="panel panel-primary" id="panels">
    <div class="panel-heading">Cuisine
    </div>
    <% @cuisines.each do |c| %>
<p><%= c.name%></p>
<%end%>
    </div>
  </div>
<div class="col-lg-9">
  <%= yield %>
</div>
</div>
```

The screenshot shows the 'curry nation' website interface. At the top, there's a navigation bar with 'curry nation' logo, 'Recipes' link, user info 'saurabh.a.bhada@gmail.com', and 'Logout' button. Below the navigation is a table displaying two recipe entries:

Title	Preference	Type	Cuisine	Servings	Time	Difficulty	Ingredients	Procedure
Double Cheese Nachos	Vegan		Mexican	3		Hard	test	test
Double Cheese Nachos	Vegetarian		Italian	3		Medium	test	test

Below the table are three sidebar filters:

- Food Preferences:** Vegetarian, Vegan, Meat, Dairy.
- Food Type:** Curry, Dessert, Sides, Breakfast.
- Cuisine:** Italian, Mexican, Indian, Chinese.

A 'Create a Recipe' button is located at the bottom left of the main content area.

Objective complete – mini debriefing

We have our completely styled page at the end of this iteration with the help of Bootstrap 3. We will use Bootstrap throughout our book and see many facets of it in the coming applications. However, this is a good start, as the first step in styling always belongs to HTML elements.

Bootstrap 3 uses a flat UI design, which is the latest trend in web designing. Also, it is not backward compatible with earlier versions.

Mission accomplished

We have created a simple recipe-sharing application by using the default Rails' methods and looked at the basics of testing. Practically, these websites can work like multiuser blogs similar to a WordPress installation, meant only for creating recipes.

Hotshot challenges

Now that we have seen how to create a simple, social recipe-sharing application, it is time to try out some challenges on our own:

- ▶ Filter recipes by clicking on food preferences, food type, cuisine, and display the results.
- ▶ Write an `ActiveRecord` query for recipe finders.
- ▶ Create another filter based on the level of difficulty, cuisine, food type, and food preferences. Also, create a radio-button field for each level of difficulty.
- ▶ Use Bootstrap to style the radio-button fields.

Project 2

Conference and Event RSVP Management

In the past 4 to 5 years, the number of events have increased manifold. This is due to the mushrooming of several different types of user groups around the world. Also, gatherings of people with similar interests are becoming commonplace nowadays. People with similar interests, for example, biking, food, movies, and blogging, also meet up and discuss topics of their interests.

Mission briefing

In this project, we will create an event and an RSVP creation website. Users of this application can sign up, log in, and create events. Once logged in, users can create, edit, and join events by creating an RSVP for it. Other users can also join events created by other users. The event can be edited only by the event owner.

We will also create a simple admin functionality where we can edit and delete these events. We will allow the admin users to approve or reject users who want to join the events. In the home page, we will have a system-wide feed of the recent events. When a user logs in, he or she will see the edit and delete options in front of the events they have created.

Conference and Event RSVP Management

Also, the users will have a section called **My Events**, as shown in the following screenshot, where they can manage all events and RSVPs in one place:

The screenshot shows a web-based event management application. At the top, there is a navigation bar with the logo 'eventst^{ाग्रह}' (eventst^{Aggregation}), followed by tabs for 'Events' and 'My Events'. On the right side of the header are links for 'saurabh.a.bhatia@gmail.com' and 'Logout'. Below the header, there are two main sections: 'Recently Created Events' and 'Search Tags'.

Recently Created Events:

- Hsinchu Ruby Meetup**
Start Date: 08/18/2013, End Date: 08/18/2013
Location: Stanley's Steak, Hsinchu
Address: Zhubei City
Agenda: Ruby Meetup
Organized By:  saurabh.a.bhatia@gmail.com
Tags: code, ruby, meetup, rails, casual, taiwan
Buttons: Show Details, Edit, Delete
- Zhubei Ruby Meetup**
Start Date: 08/18/2013, End Date: 08/18/2013
Location: Shereton Hotel, Zhubei
Address: Shereton Hotel
Agenda: Speak on Ruby
Organized By:  saurabh@rulingcom.com
Tags: ruby
Buttons: Show Details, Create an Event

Search Tags:

- code ruby meetup rails casual taiwan

Why is it awesome?

Meet-ups are a great way to meet people with similar interests. The Internet has been a catalyst in bringing people together beyond boundaries, grabbing the attention of entrepreneurs. A number of websites enable people to create events and also allow them to register for events for free or for a fee. In a way, we are enabling people to easily organize gatherings, share, and enjoy together. They might end up making friends and having a lot of fun.

We will look at various features such as tagging and tag-based search, go further into ActiveRecord migrations, creating search-friendly URLs, adding states to objects, and using class methods.

Your Hotshot objectives

While building this application, we will go through the following tasks:

- ▶ Creating and administrating events
- ▶ Creating search-friendly URLs for events
- ▶ Adding tags to events
- ▶ Tagging-based search and tag cloud
- ▶ Adding Gravatar for a user
- ▶ Creating RSVPs for events
- ▶ Adding event moderation
- ▶ Creating "My events" to manage events created by users

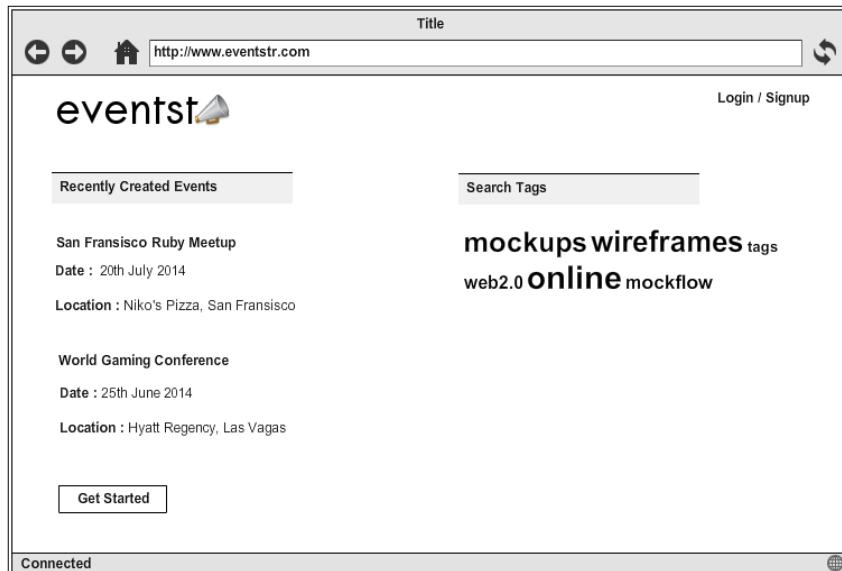
Mission checklist

We need the following software installed on the system before we start with our mission:

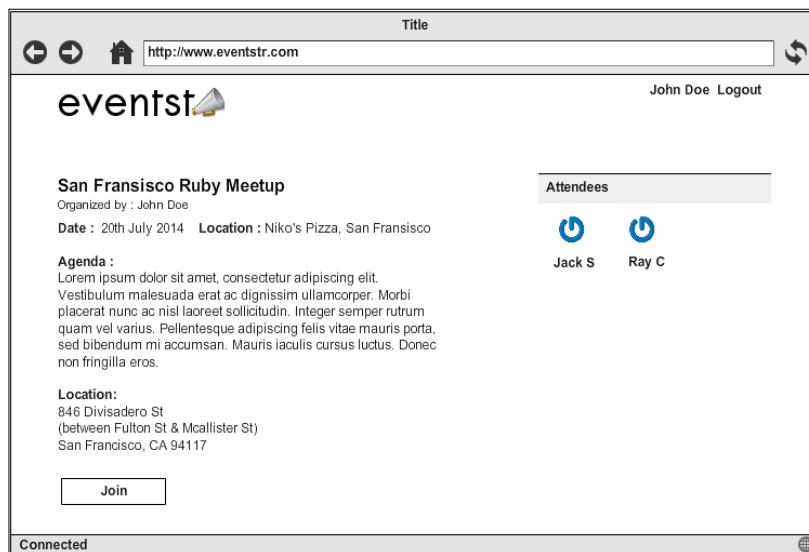
- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MySQL 6
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Sublime Text
- ▶ Devise
- ▶ Git
- ▶ A tool for mockups

Creating and administrating events

Before we begin developing our application, we will take a cue from our previous project and build mockups for our events before we start. Again, we will use MockFlow for our purpose and build it.



Also, we will create a mockup for the event page, as shown in the following screenshot:



In this task, we will look at customizing our event views and also adding the custom `before_filter` object to protect our events.

Prepare for lift off

Taking a cue from the previous project, add a scaffold for events. The events schema looks as follows:

```
create_table "events", force: true do |t|
  t.string    "title"
  t.datetime "start_date"
  t.datetime "end_date"
  t.string    "location"
  t.text      "agenda"
  t.text      "address"
  t.integer   "organizer_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end
```

Add devise gem and generate authentication methods for the application.

We will have to associate the user and event; however, the trick here is we will have multiple associations between them. Hence, we create an association with a different name as follows:

```
app/models/event.rb
class Event < ActiveRecord::Base
  belongs_to :organizers, class_name: "User"
end

app/models/user.rb
class User < ActiveRecord::Base
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
  has_many :organized_events, class_name: "Event", foreign_key:
    "organizer_id"
end
```

In order to pass the `organizer_id` object, we will use our `create` method, as shown in the following code:

```
app/controllers/events_controller.rb
def create
  @event = current_user.organized_events.new(event_params)
```

```
respond_to do |format|
  if @event.save
    format.html { redirect_to @event, notice: 'Event was
      successfully created.' }
    format.json { render action: 'show', status: :created,
      location: @event }
  else
    format.html { render action: 'new' }
    format.json { render json: @event.errors, status:
      :unprocessable_entity }
  end
end
end
```

However, as this method depends on the `current_user` object, we will add a `before_filter` object in the following code snippet to allow only the logged in users to create an event:

```
app/controllers/events_controller.rb
before_filter :authenticate_user!
```

The following screenshot shows how our form should look when we begin this task:

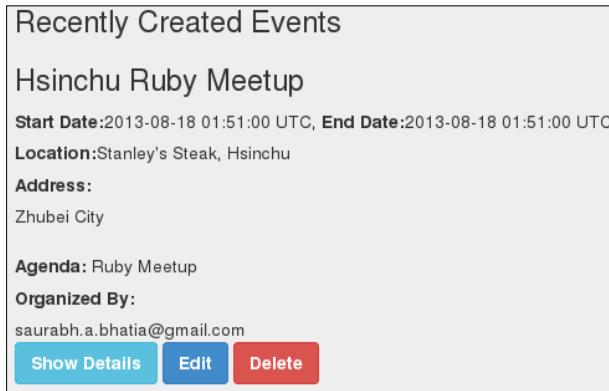
The screenshot shows a web form titled "Create an Event". The form fields include:

- Title**: An input field.
- Start date**: A date and time picker showing "2013 August 18 23:31".
- End date**: A date and time picker showing "2013 August 18 23:31".
- Location**: An input field.
- Agenda**: An input field.
- Address**: An input field.
- Create Event**: A button at the bottom left.
- Back**: A link at the bottom right.

Engage thrusters

In this task, we will customize our event date formats by performing the following steps:

1. We will start by customizing the date formats. The default format of date is `datetime` in Rails; hence, the date is displayed as the date and time function, as shown in the following screenshot:



2. We will customize these events using the `strftime` function available in Ruby for converting the date to a more human-readable format in the following manner:

```
Config/locales/en.yml  
en:  
  time:  
    formats:  
      date_format: "%m/%d/%Y"
```

```
app/views/index.html.erb  
  
<%= l event.start_date, :format => :date_format %>  
  
<%= l event.start_time, :format => :date_format %>
```

This will convert `datetime` to the date in the format of MM/DD/YYYY, as shown in the following screenshot:



3. Now that dates are formatted, we have a complete event creation and display format with us. However, because it's a Web 2.0 system, a lot of users will log in. We need to protect the events created by particular users and allow only these users to update or delete them, as we want to allow only the event owner to edit and delete the event. In order to do so, we will first add a `before_filter` method. This method has to be private to keep it protected and has to be called inside the same controller, as shown in the following code snippet:

```
app/controllers/events_controller.rb
private
  def event_owner!

    authenticate_user!
    if @event.organizer_id != current_user.id
      redirect_to events_path
      flash[:notice] = 'You do not have enough
                       permissions to do this'
    end
  end
```

This will make a method called `event_owner` available for the events controller. This method will authenticate users using devise and check if the current user ID is the same as the organizer ID of the event. If yes, then it will allow the edit, else it will redirect to all events path. Now that we will have the `before_filter event_owner!` method in place, we will protect specific methods as follows:

```
before_action :event_owner!, only:
  [:edit, :update, :destroy]
```

This will restrict the `edit`, `update`, and `destroy` methods so that it can be accessed only by the event owner.

Objective complete – mini debriefing

In the preceding task, we added a custom method to authenticate our specific `edit`, `update`, and `destroy` methods so that only the event owners can do that. This is the kind of admin facility available only to the event owners. The `before_action` method is a new way to write `before_filter` in Rails 3.2. The main functionality of `before_action` is the same; however, both can be used based on the context:

```
before_action :event_owner!, only: [:edit, :update, :destroy]
```

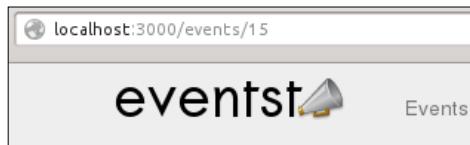
We also customized the date format to a more human readable format than the default `datetime` format of Rails. We defined the format in our locale file and called it in the view directly.

```
<%= l event.start_date, :format => :date_format %>
```

This will allow dates to be localized. Apart from this, there are multiple ways to set the `datetime` format. We can also define the date format inside our helper and call it in the view. We will continue to customize our events in the following tasks.

Creating search-friendly URLs for events

A lot of libraries have now become standard in Rails since the past few versions. Their development and maintenance activity have also caught up with various versions of Ruby and Rails, as there is a lot of community backing these libraries too. A `FriendlyId` gem is one of the most standard libraries for creating search-friendly URLs also known as **slugs**. It is highly extensible and customizable.



Engage thrusters

We will go ahead and add slugs to our application in this task:

1. We will add a `friendly_id` gem and migrations now, as follows:

```
gem 'friendly_id', '5.0.0.beta1'
```



Only Version 5.0.0 or above is compatible with Rails 4.

2. We will first add the `friendly_id` gem to the Gemfile and bundle it. Once done, we will have to first set up the migrations for slugs. For every model we have, we need to have a column for maintaining slugs:

```
:~/evenstr$ rails g migration add_slug_to_events slug:string  
invoke  active_record  
create    db/migrate/20130811083714_add_slug_to_events.rb  
  
:~/evenstr$ rake db:migrate  
==  AddSlugToEvents: migrating =====  
=====  
-- add_column(:events, :slug, :string)  
 -> 0.2797s  
==  AddSlugToEvents: migrated (0.2799s) =====  
=====
```

3. We will then enable slug creation on a model. After adding a column for slugs, we need to enable slugging in the model as follows:

```
extend FriendlyId  
friendly_id :title, use: :slugged
```

This will create a slug based on the title of the event. However, if the slug history feature is not enabled, the old URLs will give 404s. In order to maintain the old URLs intact, we need to enable history.

4. In this step, we will enable the history option for slugs.

In case we need to set up the history and version feature for slugs, we need to generate another migration. This helps us to maintain slugs and their history in a different table and still keeping it unique. This is in case we want to allow the user to edit the URL and still want to resolve the old URLs. This is particularly helpful for use cases such as blogs where a lot of URLs are bookmarked frequently:

```
:~/eventstr$ rails generate friendly_id  
rwub@rwub:~/eventstr$ rake db:migrate  
==  CreateFriendlyIdsSlugs: migrating =====  
=====  
-- create_table(:friendly_id_slugs)  
 -> 0.1736s  
-- add_index(:friendly_id_slugs, :slugable_id)  
 -> 0.1894s  
-- add_index(:friendly_id_slugs, [:slug, :slugable_type])  
 -> 0.1778s
```

```
-- add_index(:friendly_id_slugs, [:slug, :sluggable_type, :scope],  
{:unique=>true})  
-> 0.1669s  
-- add_index(:friendly_id_slugs, :sluggable_type)  
-> 0.1451s  
== CreateFriendlyIdSlugs: migrated (0.8537s) =====  
=====
```

5. If there are some existing records already, we would like to create slugs for them.

Now, as soon as we create events, we get a generated slug for our new events. In case we have events created before adding this functionality, we would have to manually update the slugs. Fire up the console and run the method to save the slugs:

```
1.9.3-p327 :001 > User.find_each(&:save)  
User Load (0.4ms)  SELECT `users`.* FROM `users` ORDER BY  
`users`.`id` ASC LIMIT 1000  
 (0.2ms)  BEGIN  
 (0.3ms)  COMMIT  
=> nil
```

This will create slugs for all the events that were previously created.

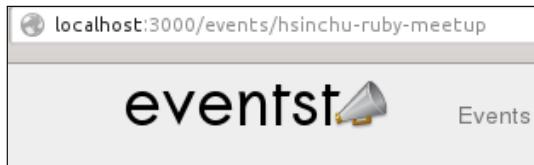
Objective complete – mini debriefing

We created functionalities for creating slugs and also gave users the option to keep a history of slugs. Slugs in the preceding task are created from the attribute(s) defined in our case's title. These slugs are created because it is easier to get crawled by search engines and provide better visibility, and in turn user experience to the app user. We used the FriendlyId Version 5 to create slugs. The friendly_id Version 5 is not compatible with the earlier versions. Two major changes in this version are as follows:

- ▶ Slugs do not get updated on the `update` method. In order to do this, we need to pass a `nil` value.
- ▶ There are multiple options to create slugs in case one of them is not unique. As shown in the following code, if a slug created with title is not unique, the `friendly_id` gem will combine title and location to make a slug:

```
friendly_id :slug_candidates, use: :slugged  
  
def slug_candidates  
[  
  :title,  
  [:title, :location],  
]  
end
```

The following screenshot shows how a typical slug looks:



Adding tags to events

Tags are quite an interesting way to organize the content. As opposed to categories, tags are created and assigned by the user. In this task, we will enable our application so that we can create and save tags for each event. In our case, tags work as a primary way to search and categorize content. `Acts_as_taggable` is a formidable solution to this problem; however, we will look into building our own tagging method that is similar to `acts_as_taggable`. This could be a fun challenge as we are trying to emulate the behavior of an advanced gem from scratch.

Engage thrusters

We will now learn how to create tags in the following steps:

1. Generate a tag model with a single attribute called name:

```
:~/eventstr$ rails g model tag name:string
invoke  active_record
create    db/migrate/20130818094312_create_tags.rb
create    app/models/tag.rb
invoke    test_unit
create    test/models/tag_test.rb
create    test/fixtures/tags.yml
db/migrations/create_tags.rb
class CreateTags < ActiveRecord::Migration
  def change
    create_table :tags do |t|
      t.string :name, index: true

      t.timestamps
    end
  end
end
```

Once we have a table to store tags, we would like multiple tags to be associated with an event. Hence, we need to create a join table that will handle multiple tags and events.

2. Generate a model called `taggings` to create a join on events and tags, as shown in the following code:

```
:~/eventstr$ rails g model tagging tag:belongs_to event:belongs_to
      invoke  active_record
      create    db/migrate/20130818095026_create_taggings.rb
      create    app/models/tagging.rb
      invoke  test_unit
      create    test/models/tagging_test.rb
      create    test/fixtures/taggings.yml
```

This will create a table with the following migration:

```
db/migrations/create_taggings.rb
class CreateTaggings < ActiveRecord::Migration
  def change
    create_table :taggings do |t|
      t.belongs_to :tag, index: true
      t.belongs_to :event, index: true

      t.timestamps
    end
  end
end
```

Also, it will generate the table in the backend with the respective IDs of the tag and the event.

table	
id	Field INTEGER PRIMARY KEY
tag_id	Field integer
event_id	Field integer
created_at	Field datetime
updated_at	Field datetime

The tagging model already has the association with the event and tag models:

```
belongs_to :tag
belongs_to :event
```

3. So, now we need to create an association between events and tags for them to use the ActiveRecord join. Add associations between events and tags.

4. Inside your event model, define the association with tags:

```
has_many :taggings  
has_many :tags, through: :taggings
```

5. Now, inside your tag model, define the association with events:

```
has_many :taggings  
has_many :events, through: :taggings
```

This will create a join between an event and a tag through the taggings table. We now have to write a method to create tags as a part of event creation.

6. Write a method in the event to enter the tags in a list as comma-separated values, as shown in the following code snippet. These values will be stripped and entered into the database:

```
def all_tags=(names)  
  self.tags = names.split(",").map do |t|  
    Tag.where(name: t.strip).first_or_create!  
  end  
end
```

7. Now, we need to add these values to the form because that's where we will enter the tags. The tags here are case sensitive, so "Awesome" and "awesome" will be treated as two different tags.

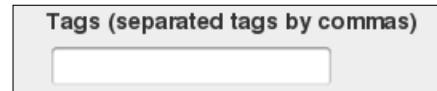
8. These tags should be mounted to a model like an attribute of it. Rails 4 has a new way to set up whitelisted attributes. Strong parameters are no longer supported and `attribute_accessor` methods. These are now defined in the controller as a private method. We will add `all_tags` as a whitelisted virtual attribute to `events_controller.rb`:

```
App/controllers/events_controller.rb  
private  
  # Never trust parameters from the scary internet, only allow  
  # the white list through.  
  def event_params  
    params.require(:event).permit(:title, :start_date,  
      :start_time, :location, :agenda, :address,  
      :organizer_id, :all_tags)  
  end
```

9. Add a list of tags to the form:

```
app/views/events/_form.html.erb
<div class="field">
  all_tags
  <%= f.label :all_tags, "List All Tags, separate each
tag by a comma" %><br />

  <%= f.text_field :all_tags %>
</div>
```



After the values are entered into the database, we will have to retrieve them and display them somewhere.

10. Retrieve the tag values from the database.

Inside the event model, add a method to call the tags by name and display them as comma-separated values:

```
app/models/event.rb
def all_tags
  tags.map(&:name).join(", ")
```

11. Display these values in the view by simply making a call on this method via the object:

```
app/views/events/show.html.erb
<%= event.all_tags %>
```

12. We can alternatively display this list using the `map` method as follows:

```
app/views/events/show.html.erb
<%= raw event.tags.map(&:name).map { |t| t }.join(', ') %>
```

Objective complete – mini debriefing

We have added tags to the events now. We used two tables, tags, and taggings to achieve this. The tags table saves the tag values, whereas the taggings table saves the association between tags and the records. We used `index:true` in our migration here. The `index:true` migration option is the same as `add_index`.

We then associated our tags and events via the taggings table. The taggings table is basically a join table to save the related values of tags and events. We then called all the tag values and displayed them as a comma-separated value list. The `first_or_create` method in ActiveRecord searches for a tag in the database. If it is not present, then a new tag is created, as shown in the following code snippet:

```
self.tags = names.split(",").map do |t|
  Tag.where(name: t.strip).first_or_create!
end
```

We had to pass tags as an attribute in the form, hence we had to add it to the attribute whitelist. Deprecation of strong parameters is a major change in Rails 4. An older way to define parameters was using `attr_accessor :all_tags`.

In Rails 3.2, parameters lead to a lot of security vulnerabilities for Rails apps; hence, it was moved to protected methods inside a controller. Also, a method called `permit` is introduced in order to create a whitelist of parameters to be allowed to pass. Only when this is done, forms will accept a certain attribute.

```
params.require(:event).permit(:title, :start_date, :start_time,
                               :location, :agenda, :address, :organizer_id, :all_tags)
```

We also saw how to add them to the views in both the form and displaying the final values. We will now go ahead and create a tag-based search and tag cloud.

The following screenshot shows how our events page looks with a list of **Tags**:

Recently Created Events

Hsinchu Ruby Meetup

Start Date: 08/18/2013 , **End Date:** 08/18/2013

Location:Stanley's Steak, Hsinchu

Address:
Zhubei City

Agenda: Ruby Meetup

Organized By:
saurabh.a.bhatia@gmail.com

Tags:
code, ruby, meetup, rails, casual, taiwan

Show Details **Edit** **Delete**

Tagging-based search and tag cloud

We will continue from our earlier task where we created tags and entered them in the database to create scope-based searches on tags. We will also count the number of times a particular tag exists and will generate the size of the tag name in the tag cloud based on this. We will do all these tasks using the class methods inside our events model.

Engage thrusters

We will now create a tag cloud from the tags that have been saved.

1. Add a tag search scope to the event controller.

Pass the value of tag as `params` and use the `tagged_with` scope to find all the records that contain a particular tag. Also, we are factoring for the records that do not have any tags associated with them:

```
app/controllers/events_controller.rb
def index
  if params[:tag]
    @events = Event.tagged_with(params[:tag])
  else
    @events = Event.all
  end
end
```

2. Add links to tags for searching.

We need to add a link to a tag and a path to search a method for tags:

```
app/views/events/index.html.erb
<%= event.all_tags.map { |t| link_to t, tag_path(t) }.join(', ') %>
```

However, we have not defined `tag_path` yet, so our next step will be to add a route.

3. Add a get route to the method where we added a search method for tag that points to the index action in the events controller:

```
config/routes.rb
get 'tags/:tag', to: 'events#index', as: :tag
```

This will pass :tag as params to the index method and generate a link to the search results page.

The screenshot shows a web page titled "Recently Created Events". Below it, a specific event is listed: "Hsinchu Ruby Meetup". The event details include:
Start Date: 08/18/2013, **End Date:** 08/18/2013
Location: Stanley's Steak, Hsinchu
Address: Zhubei City
Agenda: Ruby Meetup
Organized By: saurabh.a.bhatia@gmail.com
Tags: code, ruby, meetup, rails, casual, taiwan

A blue "Show Details" button is located at the bottom left of the event card.

4. Until now, we did not have the tagged_with method that would search the tagged events. Let's write that now in the event method:

```
app/models/event.rb
def self.tagged_with(name)
  Tag.find_by_name!(name).events
end
```

We can now list the events based on our tags:

The screenshot shows a web page titled "events" with a sub-section "Events". Below it, a list of events is shown, all of which are tagged with "taiwan". The first event listed is the same as in the previous screenshot: "Hsinchu Ruby Meetup". The event details are identical to the previous screenshot.

To generate a crowd, we need to count the number of occurrences of tags in the database.

5. Add a method to count the number of tags associated with all the events:

```
app/models/event.rb
def self.tag_counts
  Tag.select("tags.name, count(taggings.tag_id) as
  count").
  joins(:taggings).group("taggings.tag_id")
end
```

Now that we have the tag counts, we need to find a way to style them according to the sizes.

6. Add a helper method to connect this to the view for counting and rounding off in application_helper.rb, as shown in the following code:

```
app/helpers/application_helper.rb
def tag_cloud(tags, classes)
  max = tags.sort_by(&:count).last
  tags.each do |tag|
    index = tag.count.to_f / max.count * (classes.size -
    1)
    yield(tag, classes[index.round])
  end
end
```

7. Add styles to different sizes of tags based on the tag count:

```
app/assets/tags.css
.css1 { font-size: 1.0em; }
.css2 { font-size: 1.2em; }
.css3 { font-size: 1.4em; }
.css4 { font-size: 1.6em; }
```

We will add this tags.css file to our application.css manifest file:

```
app/assets/stylesheets/application.css
```

```
*= require tags
```

Finally, we will display the tags and apply the CSS classes to them.

8. Create a loop by calling the `tag_counts` method on the event model. We will also pass an array of CSS classes based on the tag count to resize our text. This loop will identify the number of times a tag appears in a model class and applies the CSS class accordingly:

```
app/views/events/index.html.erb
<div class="col-lg-4">
  <h3>Search Tags</h3>
  <div>
    <% tag_cloud Event.tag_counts, %w{css1 css2 css3
      css4} do |tag, css_class| %>
      <%= link_to tag.name, tag_path(tag.name), class:
        css_class %>
    <% end %>
  </div>
</div>
```

Objective complete – mini debriefing

In the preceding task we saw how to create a tag-based search and tag cloud. In order to create our tag cloud, we created a CSS-based tag cloud, where the size of the tag term will be determined by that tag's number of occurrences in the database.

We then created the `tagged_with` method in order to search events with a particular tag, as shown in the following code snippet:

```
def self.tagged_with(name)
  Tag.find_by_name!(name).events
end
```

In order to count the tags and generate a tag cloud, we wrote the following code snippet:

```
def tag_cloud(tags, classes)
  max = tags.sort_by(&:count).last
  tags.each do |tag|
    index = tag.count.to_f / max.count * (classes.size - 1)
    yield(tag, classes[index.round])
  end
end
```

The `tags` variable returns an array. We run the `sort_by(&:count)` method, which counts the number of occurrences of each tag, sorts the tags, and places the max-counted tag in the `max` variable. We then matched this max value of tag count with the value of the other tags; `classes` pass the value of the event in this case. The `index` variable includes the relative value of popularity of each tag in proportion to the other tags. The `yield` method finally returns the value of the tag and the popularity index of each tag.

We can refactor these methods to a concern and pass any class we want. Finally, we applied a style to the tag cloud according to the tag count, as shown in the following code snippet:

```
<% tag_cloud Event.tag_counts, %w{css1 css2 css3 css4} do |tag,
  css_class| %>
<%= link_to tag.name, tag_path(tag.name), class: css_class
%>
<% end %>
```

We get the following output:



For tags whose related events have been deleted, the tags will still remain but their taggings will be deleted. The ones with no events will generate a zero-search result.

Adding Gravatar for a user

Gravatar (also known as globally recognized avatar) is an avatar system where you can register your e-mail and upload your avatar image accordingly. This image is automatically displayed based on the e-mail on the websites where the gravatar is available. We will add a facility to automatically display the gravatar image based on the user ID.

Engage thrusters

1. Add a helper method to make a call on the gravatar method using the user's e-mail, as shown in the following code snippet:

```
app/helpers/application_helpers.rb
def avatar_url(user)
  gravatar_id = Digest::MD5::hexdigest(user.email).downcase
  "http://gravatar.com/avatar/#{gravatar_id}.png"
end
```

2. Add a method in the event model to find the event owner. We need this to display the gravatar of the event owner:

```
app/models/event.rb
def self.event_owner(organizer_id)
  User.find_by id: organizer_id
end
```

3. In the controller, make a call on this method to call the event owner:

```
app/controllers/events_controller.rb
def show
  @event_owners = @event.organizers
end
```

4. Call the helper method in the view to display the gravatar:

```
app/views/events/show.html.erb
<label>Organized By:</label><br/>
<% @event_owners.each do |event_owner| %>
<%= image_tag avatar_url(event_owner) %>
<%= event_owner.email %>
<br/>
```

Objective complete – mini debriefing

We have successfully added the gravatar image system to our application and displayed avatars of the users now. We used the gravatar API to search for the gravatar ID according to the user e-mail, as shown in the following code snippet:

```
def avatar_url(user)
  gravatar_id = Digest::MD5::hexdigest(user.email).downcase
  "http://gravatar.com/avatar/#{gravatar_id}.png"
end
```

The gravatar ID accepts an e-mail as a unique field; hence, if the e-mail address of the person is present on the gravatar site, then it will return a gravatar ID. The image can be directly browsed from a direct link to the gravatar ID.

Recently Created Events

Hsinchu Ruby Meetup

Start Date: 08/18/2013 , **End Date:** 08/18/2013
Location:Stanley's Steak, Hsinchu
Address:
Zhubei City

Agenda: Ruby Meetup
Organized By:
 saurabh.a.bhatia@gmail.com

Tags: code, ruby, meetup, rails, casual, taiwan

Show Details **Edit** **Delete**

Creating RSVPs for events

Now we have events and they can also be searched. In order to create RSVPs, we need to allow requests for joining events. In this task, we will allow users to make a request to join an event. This will generate a list of users joining a particular event. This is quite helpful in many ways. We will have to create a model called attendance where we will make a join of events and users. This is because we want to allow many users to join many events.

Engage thrusters

The following steps are performed to create RSVPs for events:

1. Create a model for attendance with the user ID and event ID. This is a simple join model, and we generate it as follows:

```
:~/eventstr$ rails g model attendance user_id:integer event_id:integer
invoke active_record
  create db/migrate/20130818045351_create_attendances.rb
  create app/models/attendance.rb

invoke test_unit
  create test/models/attendance_test.rb
```

```
create      test/fixtures/attendances.yml

:~/eventstr$ rake db:migrate
==  CreateAttendances: migrating =====
=====
-- create_table(:attendances)

-> 0.1784s

==  CreateAttendances: migrated (0.1786s) =====
=====
```

2. Add associations for the user and event in the attendance model:

```
app/models/attendance.rb
belongs_to :event
belongs_to :user
```

Do the same respectively for the event and user model too. So, in our user model, we can add:

```
app/models/user.rb
has_many :attendances
has_many :events, :through => :attendances
```

And the event model looks as follows:

```
app/models/event.rb
has_many :attendances
has_many :users, :through => :attendances
```

3. In the controller, add a method to create attendance and pass event ID and user ID as params, as shown in the following code:

```
app/models/attendance.rb
def self.join_event(user_id, event_id, state)
  self.create(user_id: user_id, event_id: event_id,
             state: state)
end
app/controllers/events_controller.rb
def join
  @attendance = Attendance.join_event(current_user.id,
    params[:event_id], 'request_sent')
  'Request Sent' if @attendance.save
  respond_with @attendance
end
```

4. Add a link to the Join event for a user to click and send the join request:

```
app/views/events/show.html.erb
<%= link_to "Join", event_join_path(:event_id =>
@event.id), :class=>"btn btn-success btn-small" %>
```

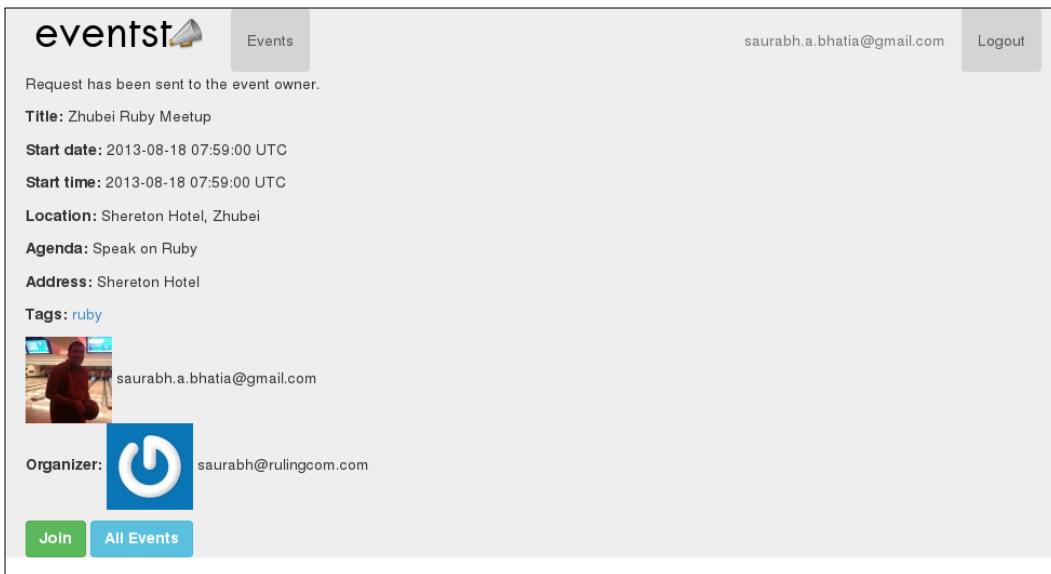
This, of course, is not complete without a route.

5. Now, we will go ahead and add the route:

```
config/routes.rb
resources :events do
  get :join, to: 'events#join', as: 'join'
end
```

Objective complete – mini debriefing

We have now created a way for users to join events. However, in the current scenario, any user can click on **Join** and join the event, as shown in the following screenshot. This, in particular, is not a great thing as sometimes seats are limited and there is room only for relevant people in the meet-up. Hence, we need some sort of moderation for our events.



The screenshot shows a web application interface for managing events. At the top, there is a navigation bar with the logo 'eventst' and a megaphone icon, followed by links for 'Events', 'Logout', and the user's email address 'saurabh.a.bhatia@gmail.com'. Below the navigation, a message says 'Request has been sent to the event owner.' The event details are listed as follows:

- Title:** Zhubei Ruby Meetup
- Start date:** 2013-08-18 07:59:00 UTC
- Start time:** 2013-08-18 07:59:00 UTC
- Location:** Shereton Hotel, Zhubei
- Agenda:** Speak on Ruby
- Address:** Shereton Hotel
- Tags:** ruby

Below the event details, there is a small profile picture of a person and the email address 'saurabh.a.bhatia@gmail.com'. Under the 'Organizer' section, there is a blue square icon with a white power symbol and the email address 'saurabh@rulingcom.com'. At the bottom of the page, there are two buttons: a green 'Join' button and a blue 'All Events' button.

Adding event moderation

Event moderation is an extremely critical feature that allows only the right kind of people to join the events. We will also look at the state machine in this task. We will use the `workflow` gem to build a state machine in order to create and manage our states.

A state machine is defined as a predefined sequence of actions that leads a process from one state to another. An event triggers the transition and changes the state.

In the classic example of a traffic signal, we have three states: stop, wait, and move. Each state is defined by a color. So, a signal turning green is a transition event, and this changes the state from stopped to moving.

In our use case, a state machine helps in the moderation process. Moderation is a multistep process, which involves the following steps:

- ▶ The user sends a request to join the event
- ▶ The event owner accepts or rejects the request

A state machine facilitates this process, where the request is an event and moderation is another event.

Engage thrusters

To add an event moderation, we will perform the following steps:

1. Add the `workflow` gem and bundle:

```
gem 'workflow', :github => 'geekq/workflow'
```

2. Add a column called `state` in the attendance table to save the current states of our users:

```
:~/eventstr$ rails g migration add_state_to_attendance
state:string
  invoke  active_record
    create    db/migrate/20130818052628_add_state_to_attendance.
rb
:~/eventstr$ rake db:migrate
==  AddStateToAttendance: migrating =====
=====
-- add_column(:attendances, :state, :string)
 -> 0.2810s
==  AddStateToAttendance: migrated (0.2812s) =====
=====
```

We need to define the states and transitions in order to accept and reject the requests.

3. Include the `workflow` method in the attendance model and inherit it from the gem. We will create a column called state in our attendance model:

```
app/models/attendance.rb
include Workflow
  workflow_column :state
```

4. Define states in the attendance model:

```
app/models/attendance.rb
workflow do
  state :request_sent do
    event :accept, :transitions_to => :accepted
    event :reject, :transitions_to => :rejected
  end
  state :accepted
  state :rejected
end
```

However, we need to persist the initial state, that is, `request_sent`. We will do this by saving it with the `join` method.

5. Persist the state on the `join` method. This will allow the user to send the request by clicking on the join link:

```
app/controllers/events_controller.rb
def join
  @attendance = Attendance.join_event(current_user.id,
  params[:event_id], 'request_sent')
end
)
```

In order to accept or reject the user, we need to toggle this state value to `accept` or `reject`.

6. Toggle the state of the user attendance for accept and reject:

```
app/controllers/events_controller.rb
before_action :set_event, only: [:show, :edit, :update,
:destroy, :accept_request, :reject_request]
def accept_request

  @attendance = Attendance.find_by(id:
  params[:attendance_id]) rescue nil
  @attendance.accept!
```

```
'Applicant Accepted' if @attendance.save
respond_with(@attendance)

end

def reject_request

@attendance =
Attendance.where(params[:attendance_id]) rescue nil
@attendance.reject!
'Applicant Rejected' if @attendance.save
@respond_with(@attendance)
end
```

Accepting and rejecting the events will also have to be wired to the views.

7. Add routes to toggle the event states:

```
config/routes.rb
resources :events do
  get :join, to: 'events#join', as: 'join'
  get :accept_request, to: 'events#accept_request', as:
'request'
  get :reject_request, to: 'events#reject_request', as:
'reject'
end
```

8. Display the requests for the event owner. We will do this by adding a class method in the event model:

```
app/views/event.rb
def self.pending_requests(event_id)
  Attendance.where(event_id: event_id, state:
'request')
end
```

We will now display them in the view:

```
app/views/events/show.html.erb
<% if user_signed_in? && @event.organizer_id ==
current_user.id%>
<label>Join Requests</label>
<% if @pending_requests.present? %>
<% @pending_requests.each do |p|%>
<%= image_tag avatar_url(p.user) %>
<%= p.user.email%>
<%= link_to 'Accept',
event_accept_request_path(:event_id => @event.id,
:attendance_id => p.id), :class=>"btn btn-success
btn-small" %>
```

```
<%= link_to 'Reject',
  event_reject_request_path(:event_id => @event.id,
  :attendance_id => p.id), :class=>"btn btn-danger
  btn-small" %>
<%end%>
<%else%>
  <p>No Pending Requests for this event</p>
<%end%>
<%end%>
```

The screenshot shows a web application interface for managing events. At the top, there's a navigation bar with 'Events' and user information 'saurabh@rulingcom.com' and 'Logout'. The main content area displays a message 'Signed in successfully.' Below it, event details are listed: Title: Zhubei Ruby Meetup, Start date: 2013-08-18 07:59:00 UTC, Start time: 2013-08-18 07:59:00 UTC, Location: Shereton Hotel, Zhubei, Agenda: Speak on Ruby, Address: Shereton Hotel, and Tags: ruby. A thumbnail image of a person is shown next to the email address saurabh.a.bhatia@gmail.com, with 'Accept' and 'Reject' buttons. Below this, the organizer information is shown: Organizer: with a blue power icon, email saurabh@rulingcom.com, and buttons for 'Join', 'All Events', 'Edit', and 'Delete'.

9. Display the accepted members.

In order to do this, we will first create a scope in our attendance model and a model class method in the event model:

```
app/model/attendance.rb
  scope :accepted, -> {where(state: 'accepted')}  
app/model/
    def self.show_accepted_attendees(event_id)
      Attendance.accepted.where(event_id: event_id)
    end
```

Call this scope in the controller:

```
@attendees = Event.show_accepted_attendees(@event.id)
```

Display the attendees on the event page:

```
<label>Attendees</label>
<% @attendees.each do |a| %>
  <%= image_tag avatar_url(a.user) %>
  <%= a.user.email%>
<%end%>
```

Objective complete – mini debriefing

We have created methods to accept and reject the event membership based on moderation. We used the workflow gem to do so. We defined the states and events in the attendance model:

```
workflow do
state :request_sent do
  event :accept, :transitions_to => :accepted
  event :reject, :transitions_to => :rejected
end
state :accepted
state :rejected
end
```

When we created an attendance, that is, when a user clicks on the **Join** button, we wrote a method to create a new attendance with `event_id`, `user_id`, and the state. We associated attendance with events and users. This is because we need to track who wants to attend which event. In order to toggle the state, we just need to call the transition event on the object. This will update the attendance column of the state as follows:

```
@attendance.accept!
```

We have also displayed the accepted members and hence we have a confirmed attendee list. In order to do this, we created scopes according to different states:

```
scope :accepted, -> {where(state: 'accepted')}
```

We made a call on this scope and passed `event_id` in the argument to find all the attendances who have been accepted by the moderator for a particular event:

```
Attendance.accepted.where(event_id: event_id)
```

Likewise, we have also defined a scope for the members whose moderation is pending:

```
scope :pending, -> {where(state: 'request_sent')}
```

We made a query on the scope in our controller:

```
Attendance.pending.where(event_id: @event.id)
```

Finally, the pending requests were displayed in our view using the loop:

```
<% if @pending_request.present? %>
<% @pending_requests.each do |p| %>
  <%= image_tag avatar_url(p.user) %>
  <%= p.user.email%>
```

In order to accept the request, we passed `event_id` and `attendance_id` to the `accept` method in our events controller. This method will toggle the state of our attendance to `accepted` and likewise for `rejected`. Here, `p.id` is the the ID of the attendance whose state is `request_sent` and confirmation is pending:

```
<%= link_to 'Accept',
  event_accept_request_path(:event_id => @event.id,
    :attendance_id => p.id), :class=>"btn btn-success
    btn-small" %>
<%= link_to 'Reject',
  event_reject_request_path(:event_id => @event.id,
    :attendance_id => p.id), :class=>"btn btn-danger
    btn-small" %>
<%end%>
```

The accepted members of an event are shown in the following screenshot:

The screenshot shows a web application interface for managing events. At the top, there's a header with the logo 'eventst' and a megaphone icon, followed by 'Events', the user's email 'saurabh.a.bhatia@gmail.com', and a 'Logout' button.

The main content area displays event details:

- Title:** Hsinchu Ruby Meetup
- Start date:** 2013-08-18 01:51:00 UTC
- Start time:** 2013-08-18 01:51:00 UTC
- Location:** Stanley's Steak, Hsinchu
- Agenda:** Ruby Meetup
- Address:** Zhubei City
- Tags:** code, ruby, meetup

A section titled 'Join Requests' indicates 'No Pending Requests for this event'.

Below, there are sections for 'Attendees' and 'Organizer'. The 'Attendees' section shows a blue placeholder icon with a white power symbol, the email 'saurabh@rulingcom.com', and a small thumbnail image of a person. The 'Organizer' section shows a thumbnail image of a person and the email 'saurabh.a.bhatia@gmail.com'.

At the bottom, there are four buttons: 'Join' (green), 'All Events' (blue), 'Edit' (blue), and 'Delete' (red).

Creating "My events" to manage events created by users

"My events" is a task where a user can see all the events created by him/her. This is sometimes very critical as the system feed is shown in the general list, and it is not convenient for a user to search for his events every time he/she needs to edit it. This task will provide them with all those events under one tag, and hence it is very convenient for them.

Engage thrusters

We will now create a separate section to display events created by a particular user:

1. We already have an association between users and events:

```
app/models/event.rb
  belongs_to :organizers, class_name: "User"
app/models/user.rb
  has_many :organized_events, class_name: "Event",
    foreign_key: "organizer_id"
```

We will retrieve all the events organized by a user via the association, but we need to make a call on this method in the controller.

2. So, we go ahead and make a call on it by passing the current user ID as a parameter to it in the events controller:

```
App/controllers/events_controller.rb
  def my_events
    @events = current_user.organized_events
  end
```

Now, we have the controller method in place, so we need to display this task somewhere.

3. What we need is a view for `my_events` under the events views. We will create a blank file called `my_events.html.erb`. The code in `my_events` will essentially be the same as `index.html.erb` because even here we are making a list of events. We will also refactor tag cloud into a different partial:

```
app/views/events/_tag_cloud.html.erb
<!-- Displaying the tag cloud div-->
```

```
<div class="col-lg-4">
  <h3>Search Tags</h3>
  <div>
```

```
<!-- Generating the tag cloud -->

<% tag_cloud Event.tag_counts, %w{css1 css2 css3 css4}>
  do |tag, css_class| %>
    <%= link_to tag.name, tag_path(tag.name), class:
      css_class %>
    <% end %>
  </div>
</div>
</div>

app/views/events/my_events.html.erb
<div class="row">
  <div class="col-lg-8">
    <!--List Recently Created Events-->

    <h3>Recently Created Events</h3>
    <% @events.each do |event| %>
      <h3><%= event.title %></h3>
      <label>Start Date:</label><%= l event.start_date,
        :format => :date_format %>
      <label>End Date:</label><%= l event.start_time,
        :format => :date_format %><br/>
      <label>Location:</label><%= event.location
        %><br/>
      <label>Address:</label>
      <address>
        <%= event.address %><br/>
      </address>
      <label>Agenda:</label>
      <%= event.agenda %><br/>
      <label>Organized By:</label><br/>
      <%@event_owner =
        Event.event_owner(event.organizer_id)%>
      <%= image_tag avatar_url(@event_owner) %>
      <%= @event_owner.email %>
      <br/>
      <!-- Display Tags-->

      <label>Tags:</label><br/>
      <%= raw event.tags.map(&:name).map { |t| link_to
        t, tag_path(t) }.join(', ') %><br/><br/>

      <%= link_to 'Show Details', event, :class=>"btn
        btn-info btn-small" %>
```

```
<% if user_signed_in? && event.organizer_id ==  
   current_user.id%>  
  <%= link_to 'Edit', edit_event_path(event),  
             :class=>"btn btn-primary btn-small" %>  
  <%= link_to 'Delete', event, method: :delete,  
             data: { confirm: 'Are you sure?' },  
             :class=>"btn btn-danger btn-small" %>  
<%end%><br/><br/>  
<% end %>  
  
</div>  
<%= render "tag_cloud"%>  
<br>  
  
<%= link_to 'Create an Event', new_event_path,  
             :class=>"btn btn-default btn-primary" %>
```

4. Finally, wire this up using a route in `routes.rb` and we are good to go:

```
config/routes.rb  
get  :my_events, to: 'events#my_events', as: 'my_events'
```

Objective complete – mini debriefing

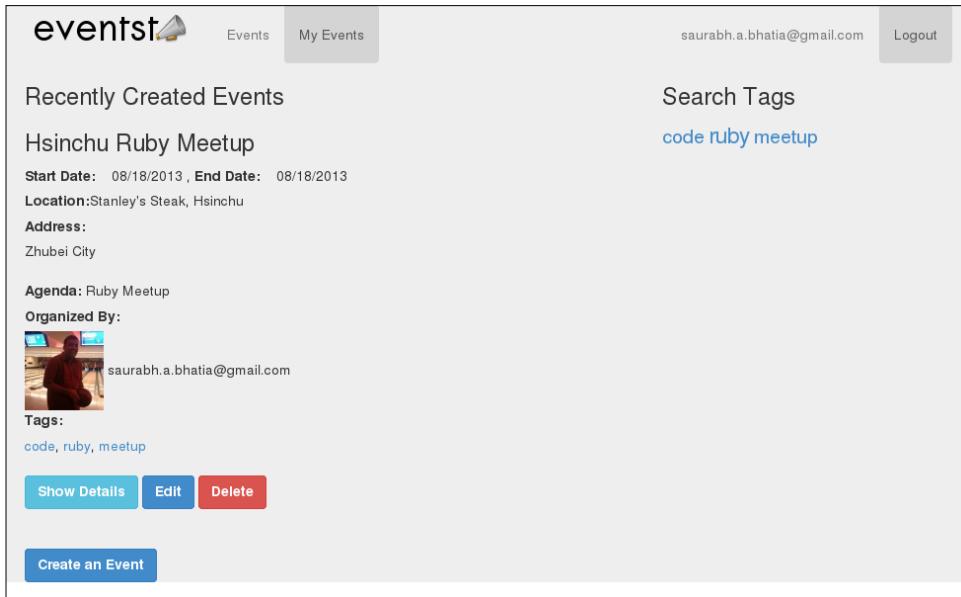
We created a separate "My events" area where a user can manage their events with ease. We did this by first creating an association between user and events. This association created a method called `user_object.organized_events` to retrieve all the events organized by a particular user. We used it in our controller by calling this on the `current_user` object:

```
current_user.organized_events
```

It is noteworthy that we have created two different types of associations between the user and event models:

- One is a named association, where we denoted users as `organizers` and events as `organized_events`. In order to identify the models, we use the attribute called `class`.
- The other is a `has_many :through attendance` association, where we made a join table to manage attendees for a particular event.

We also added a navigation called "My events", which is only visible once the user logs in to the system, as shown in the following screenshot:



Mission accomplished

We have successfully created an event RSVP application, where users can create, administer, and moderate their events. Other users can send requests to join the events. We looked at various concepts such as tagging, tag-based search, tag cloud, and gravatar during the course of this project. Some of the topics we broadly covered in this project are as follows:

- ▶ Creating multiple associations between the same models
- ▶ Adding named associations and the `has_many :through` association
- ▶ Using the `friendly_id` gem to create slugs for each user
- ▶ Creating tags for each event
- ▶ Counting the tags and creating a tag cloud from it
- ▶ Adding a state machine in order to create RSVP for an event
- ▶ Displaying the gravatar of a user
- ▶ Creating joins, usage of scopes, chained queries with scope, and additional conditions

Hotshot challenges

We have a few exercises to look into at the end of this project:

- ▶ Setting the visibility of the **Join events** button should not be visible to the owners of the events
- ▶ Restricting the owners of the event to join the event
- ▶ Creating a tag-based search in a textbox
- ▶ Displaying similar events for users based on tags
- ▶ Adding validations and tests for the entire application

Project 3

Creating an Online Social Pinboard

Every now and then, there are changes that alter our perspective of how we do things. One of these things is pinterest.com. The idea of an online **pinboard** to collect what we like is so appealing. It is a great way to organize personal information. For entrepreneurs, it gives direct insight into the likes and dislikes of a consumer. Hence, pinboards have gained importance and are now becoming specific to interests.

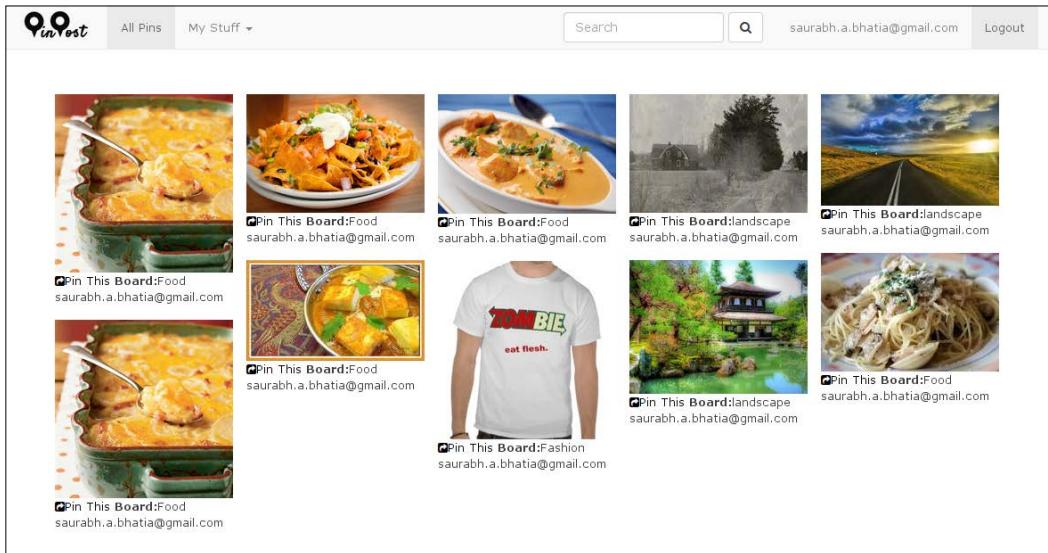
Mission briefing

We will create an online pinboard where users can collect and pin up what they like by uploading pictures. These pictures can also pinned by other users on their own pinboards.

During the course of this project, we will work with some popular jQuery plugins that have common use cases. The grid layout, infinite scroll, and modal box are some of the plugins we will look at. We will also create a mailer daemon that runs a job in the backend to send a weekly mail. Also, we will look at the basics of full-text searching and implement one in our app. Lastly, we will look at some tricks to prevent cross-site scripting and Rails security.

Creating an Online Social Pinboard

Our finished applications looks as shown in the following screenshot:



Why is it awesome?

An online platform to pin up things is a great way to look at the kind of fashion, food, design, and photography, among many others, that is trending. It is also a very visual medium to market one's creations. It is more effective than any textual medium as it creates direct impact on the seeker. Repinning a post also allows us to track trends related to various topics as the pins are arranged in boards by an individual's area of interest.

Your Hotshot objectives

We will have to perform the following tasks while building this application:

- ▶ Creating file uploads and image resizing
- ▶ Creating an infinitely scrollable page
- ▶ Creating a responsive grid layout
- ▶ Adding a full-text search
- ▶ Resharing the pins and creating modal boxes using jQuery
- ▶ Enabling the application to send e-mail
- ▶ Securing application from cross-site scripting or XSS

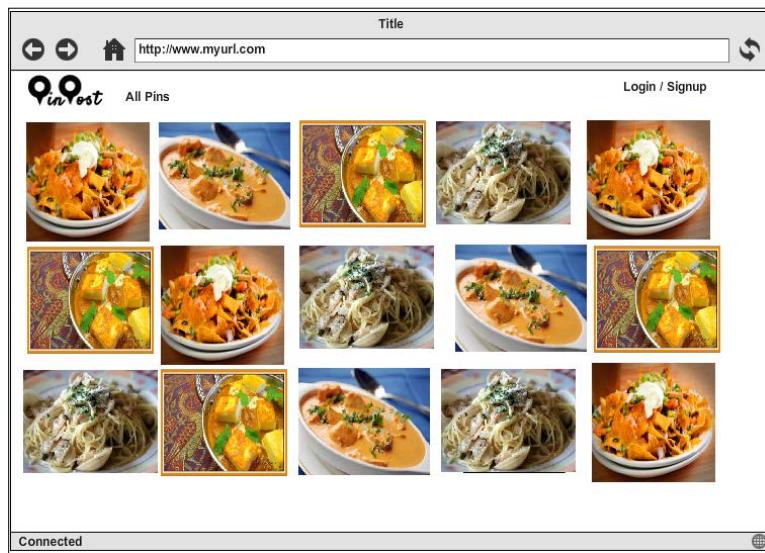
Mission checklist

We need the following installed on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MySQL 6
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Sublime Text
- ▶ Devise
- ▶ Git
- ▶ A tool for mockups
- ▶ jQuery
- ▶ ImageMagick and RMagick
- ▶ Solr

Creating file uploads and image resizing

As seen in previous projects, we will mockup our application page and create a sample layout similar to Pinterest as follows:



In this section, we will use the `carrierwave` gem to upload images and resize them into different sizes in order to display them on different pages. For example, we will display thumbnails on listing pages and larger images on individual pages.

Prepare for lift off

Before we start off with creating the upload methods, we will create two models and controllers for `board` and `pin` as follows:

```
$ rails g scaffold board title:string description:text  
$rails g scaffold pin name:string image:string board_id:integer
```

We will create an association between `pin` and `board` as follows:

```
models/pin.rb  
  belongs_to :board  
models/board.rb  
  has_many :pins
```

A user model is generated using the `devise` gem. We will also create an association between `user` and `boards`:

```
models/board.rb  
  belongs_to :user  
models/user.rb  
  has_many :boards
```

We will also use `friendly_id` to create slugs for `board` and `pin`:

```
models/board.rb  
  extend FriendlyId  
  friendly_id :title, use: :slugged  
models/pin.rb  
  extend FriendlyId  
  friendly_id :name, use: :slugged
```

`Board` is a way to organize pins, so all pins belong to a particular board. Also, these pins are a visual medium and hence full of images. So, we first need to get the images right. We will use the `carrierwave` gem to build the file uploading methods. It is a very standard method to add file uploads of all kinds.

`ImageMagick` is a dependency for our project, and we need to install it from source. Detailed installation instructions for `ImageMagick` can be found at <http://www.imagemagick.org/script/advanced-unix-installation.php>.

Once ImageMagick is installed and tested, install RMagick:

```
$ gem install rmagick
```

Engage thrusters

To create the file uploads, we will perform the following steps:

1. Add carrierwave to Gemfile and run bundle:

```
Gemfile
gem 'carrierwave'
```

2. Generate the uploader file:

```
:~/pinpost$ rails g uploader image
create  app/uploaders/image_uploader.rb
```

This will create a new folder inside the app folder called uploader and generate the file under it.

3. We will use the filesystem to store and serve files here. The files are renamed to suit the models:

```
app/uploaders/image_uploader.rb
  storage :file
  def store_dir
    Rails.env.production? ? (environment_folder = "production") :
(environment_folder = "test")
    "uploads/#{environment_folder}/#{model.class.to_s.
underscore}/#{mounted_as}/#{model.id}"
  end
```

4. These uploaders are reusable and the same one can be mounted on multiple models in our pin model:

```
app/uploaders/image_uploader.rb
  mount_uploader :image, ImageUploader
```

5. At this point, we need to add image attributes to our pin model:

```
$ rails g migration add_image_to_pins image:string
  invoke  active_record
  create    db/migrate/20140130025412_add_image_to_pins.rb
```

6. We will need to whitelist the image attributes so that they can be retrieved from a form and stored in the database.

7. Add the image parameters to the whitelist in your pins_controller file:

```
app/controllers/pins_controller.rb
def pin_params
    params.require(:pin).permit(:name, :image, :image_cache,
:board_id)
end
```

8. The carrierwave gem maps the f.file_field form helper to the carrierwave uploader method in order to upload the files. So we can add this to our form:

```
app/views/pins/show.html.erb
<div class="field">
    <%= @pin.image_url if @pin.image? %>
    <%= f.file_field :image %>
    <%= f.hidden_field :image_cache %>
</div>
```

The form to create a pin looks like what is shown in the following screenshot:

A screenshot of a web form for creating a pin. The form has a 'Name' field containing 'Pasta Au Gratin'. Below it is a 'Choose File' button with the path 'au-gratin-potatoes.jpg' displayed. A dropdown menu is open, showing 'Food' as the selected option. At the bottom of the form is a 'Create Pin' button.

9. Once the images are uploaded, we can display them.
10. In order to do so, we can directly make a call on the uploader name with a helper method called url to get the image file path:

```
app/views/pins/index.html.erb
<%=link_to(image_tag(pin.image.url, :width=>"200",
:height=>"200"), pin) %>
```

11. However, instead of manually defining width and height of the image, it's better to have them defined as a geometry and scale them during the time of upload.
12. Define geometries to resize the images to multiple sizes on different pages:

- ❑ Add the rmagick gem and install it:

```
Gemfile
gem 'rmagick'
```

- ❑ Configure it inside your uploader file:

```
app/uploaders/image_uploader.rb
include CarrierWave::RMagick
```

- Define different geometries for your image sizes:

```
app/uploaders/image_uploader.rb
# Create different versions of your uploaded files:
version :thumb do
  process :resize_to_fit => [200, 200]
end
version :normal do
  process :resize_to_fit => [350, 350]
end
```

13. Now that we have defined different sizes, we need to resize all the existing images and new ones to same sizes. In order to do so, we need a method that allows us to do this in one batch.
14. After defining the geometries, we need our already uploaded files to be resized to the specified geometries. In order to do so, we will first create a migration:

```
$ rails g migration recreate_old_thumbnails
  invoke  active_record
    create    db/migrate/20140130033618_recreate_old_thumbnails.
rb
db/migrate/20140130033618_recreate_old_thumbnails.rb
class RecreateOldThumbnails < ActiveRecord::Migration
  def up
    Pin.all.each { |p| p.image.recreate_versions! if p.image}
  end
  def down
  end
end
```

15. For the index page, modify views to call certain geometries on a certain page:

```
app/views/pins/index.html.erb
<%=link_to(image_tag(pin.image.thumb.url), pin) %>
```

16. Similarly for the show page, modify views as explained in the preceding point:

```
app/views/pins/show.html.erb
<%=image_tag @pin.image.normal.url %>
```

17. We will write a test for our uploader file as follows:

```
test/uploaders/image_uploader_test.rb
require_relative '../test_helper'
require 'rubygems'
require 'RMagick'
require 'carrierwave'
require_relative '../../../../../app/uploaders/image_uploader'
class ImageUploaderTest < MiniTest::Unit::TestCase
  FILENAME = 'well.jpeg'
  STORE_DIR = 'tmp/uploads/store'
  CACHE_DIR = 'tmp/uploads/cache'
  STORE_PATH = File.join __dir__, '..', '..', STORE_DIR
  CACHE_PATH = File.join __dir__, '..', '..', CACHE_DIR

  class ::ImageUploader
    storage :file
    def store_dir; STORE_PATH; end
    def cache_dir; CACHE_PATH; end
  end
  def setup
    @file = File.new "#{__dir__}/../test_files/#{FILENAME}"
  end
  def clear_after_test
    FileUtils.rm_rf STORE_PATH
    FileUtils.rm_rf CACHE_PATH
  end
  def test_image_upload
    uploader = ImageUploader.new
    uploader.store!(@file)
    assert_equal Digest::SHA2.file(@file).hexdigest, Digest::SHA2.
      file("#{STORE_PATH}/#{FILENAME}").hexdigest
  end
  def after_tests
  end
end
```

Objective complete – mini debriefing

The carrierwave gem creates a separate folder for the upload-related code:

app/uploaders/image_uploader.rb

In many ways, it's a very clean way to keep the uploader-related code abstracted from the rest of the code. The code to upload images is reusable and maintainable:

```
storage :file
def store_dir
  "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
end
```

The storage rule defines the storage mechanism to store files. We can also use Amazon S3 or Rackspace with the help of the `fog` gem.

The `storage_dir` defines the directory where the program stores the image. It generates the directories according to the `model` class, the type of asset (image, file, and so on), and the record number.

In case the form validation fails, the `file` field is reset. For the form to remember the filename even when the validation fails, we add a `image_cache` field in the form. We also add it to the `permitted_params` in our controller.

We then create different versions of the same file during the upload. We use RMagick, which is Ruby's interface for ImageMagick, in order to read and process the image files:

```
include CarrierWave::RMagick
```

Another option is to use MiniMagick, another interface for ImageMagick, known to consume less memory than RMagick:

```
include CarrierWave::MiniMagick
```

The `version` rule in uploader helps to identify and create versions according to the defined geometry. In order to scale the image to the specified dimensions, we defined the `:resize_to_fit` method. This method will alter the dimensions of the image:

```
version :thumb do
  process :resize_to_fit => [200, 200]
end
```

In order to crop a part of the image, we can define the `:resize_to_fill` method. This will keep the dimensions of the image intact, while cropping out the defined dimensions from the image:

```
process :resize_to_fill => [200, 200]
```

In order to display the image, we accessed it via the following rule:

```
pin.image.thumb.url
```

It can be read as follows:

```
Object Name. Uploader Name. Url
```

To test our uploader, we first load all the required classes, RMagick to resize, carrierwave to upload, and our uploader class:

```
require_relative '../test_helper'  
require 'rubygems'  
require 'RMagick'  
require 'carrierwave'  
require_relative '../../app/uploaders/image_uploader'
```

We then set up all the parameters required to create the upload method:

```
class ImageUploaderTest < MiniTest::Unit::TestCase  
  FILENAME = 'well.jpeg'  
  STORE_DIR = 'tmp/uploads/store'  
  CACHE_DIR = 'tmp/uploads/cache'  
  STORE_PATH = File.join __dir__, '..', '..', STORE_DIR  
  CACHE_PATH = File.join __dir__, '..', '..', CACHE_DIR
```

We then create our storage directories:

```
class ::ImageUploader  
  storage :file  
  def store_dir; STORE_PATH; end  
  def cache_dir; CACHE_PATH; end  
end
```

We add a method to delete the directories after the upload test passes:

```
def clear_after_test  
  FileUtils.rm_rf STORE_PATH  
  FileUtils.rm_rf CACHE_PATH  
end
```

We actually send the file to upload and match by assertion depending on whether it has been uploaded:

```
def test_image_upload  
  uploader = ImageUploader.new  
  uploader.store!(@file)  
  assert_equal Digest::SHA2.file(@file).hexdigest, Digest::SHA2.  
  file("#{STORE_PATH}/#{FILENAME}").hexdigest  
end
```

Creating an infinitely scrollable page

We are creating a social website and hope to attract several users. Very soon, with the increase in data, we will have to figure out how to arrange the data in the form of pages. We will now add pagination and see how to create and fit it in the context of our website. We will first look at creating pagination using **Kaminari** as the solution. We will then convert it to an infinitely scrollable page by identifying the end of a page and rendering the next page immediately after that.

Engage thrusters

We will now create an infinitely scrollable page for our application:

1. First add the `kaminari` gem and set it up.
2. Add the `kaminari` gem to your `Gemfile` and run `bundle install`:
`gem 'kaminari'`
3. Generate the configuration file in the initializers:
`rails g kaminari:config`
4. Once it is set up, we will add the pagination methods.
5. The `kaminari` gem methods bind to models, so we need to define the `per_page` method in each model. This will define the number of records after which a new page will be generated:
`app/models/pin.rb`
`paginates_per 10`
6. In your controller, find and arrange the pins with the latest ones on the top and make a call on the `paginates_per` method.
`app/controller/pins_controller.rb`

```
def index
  @pins = Pin.order(:created_at).page(params[:page])
end
```
7. Once the pagination methods are in place, we will render these records into pages by inserting the following at the end of the page:
`app/views/pins/index.html.erb`
`<%= paginate @pins %>`
8. We now have a working pagination in our application. We will now create an infinitely scrollable page using the jQuery library called `jQuery infinitemscroll`. We will now have to generate jQuery files and add the `jQuery infinitemscroll` jQuery library to the application.

9. First generate jQuery files in the Rails public folder using the Rails jQuery generator command:

```
:~/pinpost$ rails g jquery:install11
      remove  public/javascripts/prototype.js
      remove  public/javascripts/effects.js
      remove  public/javascripts/dragdrop.js
      remove  public/javascripts/controls.js
      copying  jQuery (1.10.2)
      create   public/javascripts/jquery.js
      create   public/javascripts/jquery.min.js
      copying  jQuery UJS adapter (e9e8b8)
      remove   public/javascripts/rails.js
      create   public/javascripts/jquery_ujs.js
```

10. Once this is done, download infinite scroll (<https://github.com/paulirish/infinite-scroll>) and add the jQuery infinitescroll library to the application:

```
:~/pinpost/vendor/assets/javascripts$ ls
jquery.infinitescroll.js
```

11. Then require the infinitescroll library in application.js:

```
app/assets/application.js
//= require jquery.infinitescroll
```

12. We need this script to only run on the page where we have to display all the pins.

Thus, we add the following script to pins.js.coffee:

```
app/assets/pins.js.coffee
$(document).ready ->
  $("#posts").infinitescroll
    navSelector: "nav.pagination"
    nextSelector: "nav.pagination a[rel=next]"
    itemSelector: "#posts tr.post"
```

13. The next item selector binds to a particular tag inside your tag structure. The nav.pagination method will fire the next page to bring the next batch of records in order to display them.

14. We will now find the end of the page, generate a div element, and append it to the page.

15. For the page to look infinitely scrollable, we will have to identify the end of the page and generate a `div` element in order to display the next page. We will create an `index.js.erb` file inside our `views/pins` folder:

```
app/views/pins/index.js.erb
$( "#posts" ).append("<div class='page'><%= escape_
javascript(render(@pins)) %></div>");
```

16. Finally, we will modify the index page to display pagination.
17. We will assign an ID called `posts` to table, so the `index.js.erb` file can bind to it. Each page will have a `tbody` class `page` and each pin will bind to a `post` class:

```
app/views/pins/index.html.erb
<table id="posts">
  <tbody class="page">
    <% @pins.each do |pin| %>
      <tr class="post">
        <td> <%= link_to(image_tag(pin.image.thumb.url), pin) %><p>
          <strong>Board:</strong><%= pin.board.title %><br/><%= pin.
          board.user.email %></p></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

18. We will add a validation to make sure `title` is present. User e-mail is a mandate to create the account, so the `devise` gem has already taken care of it:

```
app/models/board.rb
validates :title, presence: true
```

19. This is all we need need to create an infinitely scrollable page.

Objective complete – mini debriefing

We just created a page with endless pagination. We looked at normal pagination that sorts several records page-wise. We used the `kaminari` gem to create the pagination inside our application. When we generate the configuration, a `kaminari_config.rb` file is generated:

```
config/initializers/kaminari_config.rb
Kaminari.configure do |config|
  # config.default_per_page = 25
  # config.max_per_page = nil
  # config.window = 4
  # config.outer_window = 0
  # config.left = 0
```

```
# config.right = 0
# config.page_method_name = :page
# config.param_name = :page
end
```

The `config.param_name` option changes the page name required for pagination. By default, it is `page`. Then, we defined the `paginates_per` method to limit the number of records to be displayed in a page:

```
paginates_per 10
```

In order to render the pagination, we add the partial call in our view:

```
<%= paginate @pins %>
```

We then looked at making these pages paginate one after another and append at the end of each page. We used JavaScript in order to create the infinite scroll. We used a combination of jQuery and CoffeeScript in order to create the infinite scroll. It is noteworthy that CoffeeScript is a language that compiles to JavaScript. So jQuery-related code or any other code related to the JavaScript framework can be written as CoffeeScript and then compiled to JavaScript. Also, it is neatly integrated with the Rails framework, so all controllers have a CoffeeScript associated with them.

There are several libraries that provide similar functionalities, for example, `sausage.js` is a simple jQuery library with similar functions. Also, Masonry and Wookmark come with in-built methods to generate infinite scrolls. We used a jQuery plugin called `jquery_infiniteScroll` in order to implement it. The plugin can be downloaded from GitHub (<http://www.infinite-scroll.com/infinite-scroll-jquery-plugin/>).

The first selector is meant for the page navigation and this will be hidden:

```
navSelector: "nav.pagination"
```

The next page is automatically identified by `nextSelector` and it looks for the next set of posts to render:

```
nextSelector: "nav.pagination a[rel=next]"
```

Also, `ItemSelector` will render the next page or the next set of posts right after the end of the page is reached:

```
itemSelector: "#posts tr.post"
```

In Rails, to bind a JavaScript to a controller method, we have to create an action-specific JavaScript file. Thus, we create `index.js.erb`. We will be able to retrieve the posts and append them to the bottom of the page:

```
$("#posts").append("<div class='page'><%= escape_javascript(render(@pins)) %></div>");
```

In order to bind the JavaScript method to HTML, we have to create a `table` (as `tr` and `td` are inside the `table` HTML attribute) and call the `post` class on it. This will make the infinite scroll method applicable to the HTML:

```
<table id="posts">
  <tbody class="page">
    <% @pins.each do |pin| %>
      <tr class="post">
        <td> <%=link_to(image_tag(pin.image.thumb.url), pin)><p>
          <strong>Board:</strong><%= pin.board.title %><br/><%= pin.
          board.user.email %></p></td>
      </tr>
    <% end %>
  </tbody>
</table>
```

Creating a responsive grid layout

One of the most eye catching features of Pinterest and several other online pinboards is the way pins are displayed. They are arranged as a grid of images alongside each other. This is one of the greatest innovations and turning points in the creation of user experience. As previously mentioned, Masonry and Wookmark are some of the libraries that generate these kind of grids.

Prepare for lift off

Download the Wookmark from its GitHub repository (<https://github.com/GBKS/Wookmark-jQuery>). Place the `jquery.wookmark.js` file in the `app/assets` folder.

Engage thrusters

We will add the Pinterest-style grid layout in this task:

1. First add the Wookmark library to the JavaScript files.
2. Add the `jquery.wookmark.js` file in the JavaScript files and require in `application.js`:
:`~/pinpost/app/assets/javascripts$ jquery.wookmark.js`
3. In `application.js`, add the following line:
`//= require jquery.wookmark`
4. Then initiate the JavaScript and generate a grid.

5. Initiate the function and bind it to the `tiles` ID. Also, bind it to a `td` so that we have all the images inside the `td`. We will also handle clicks and randomize the height of an image so that it looks like the images flow into one another. This will also help to resize images in a responsive format:

```
app/views/pins/index.html.erb
<script type="text/javascript">
    var $handler = $('#tiles td');
    $handler.wookmark({
        autoResize: true,
        container: $('#main'),
        offset: 5,
        outerOffset: 10,
        itemWidth: 210
    });
    $handler.click(function(){
        var newHeight = $('img', this).height() + Math.round(Math.
random() * 300 + 30);
        $(this).css('height', newHeight+'px');
        // Update the layout.
        $handler.wookmark();
    });
</script>
```

6. Next, we will create the `div` element with ID as `main` and call the grid inside it.
7. Create a `div` element called `main` as mentioned in the Wookmark initializer and bind `tbody` to `tiles`. The `<td>` tags under it will inherit the styles from this class:

```
app/views/pins/index.html.erb
<div id="main" role="main">
<table id="posts">
    <tbody id="tiles" class="page">
        <% @pins.each do |pin| %>
            <tr class="post">
                <td> <%= link_to(image_tag(pin.image.thumb.url), pin) %><p>
                    <strong>Board:</strong><%= pin.board.title %><br/><%=

pin.board.user.email %></p></td>
            </tr>
        <% end %>
    </tbody>
</table>
<%= paginate @pins %>
</div>
```

Objective complete – mini debriefing

In order to generate the grid layout to display all the pins, we used a jQuery plugin called `woomark.js`. We first created a variable called `handler` that binds to the `td` element of the table, which is each cell of the row:

```
var $handler = $('#tiles td');
```

Then, we defined the variables required for each cell to be generated using `woomark`. `Container` is the element based on which the width of each column is calculated. The `offset` element is used to define the distance between the two objects in a row:

```
$handler.woomark({
    autoResize: true,
    container: $('#main'),
    offset: 5,
    outerOffset: 10,
    itemWidth: 210
});
```

Then, we created an event that randomizes the event size and creates grid variable-sized images:

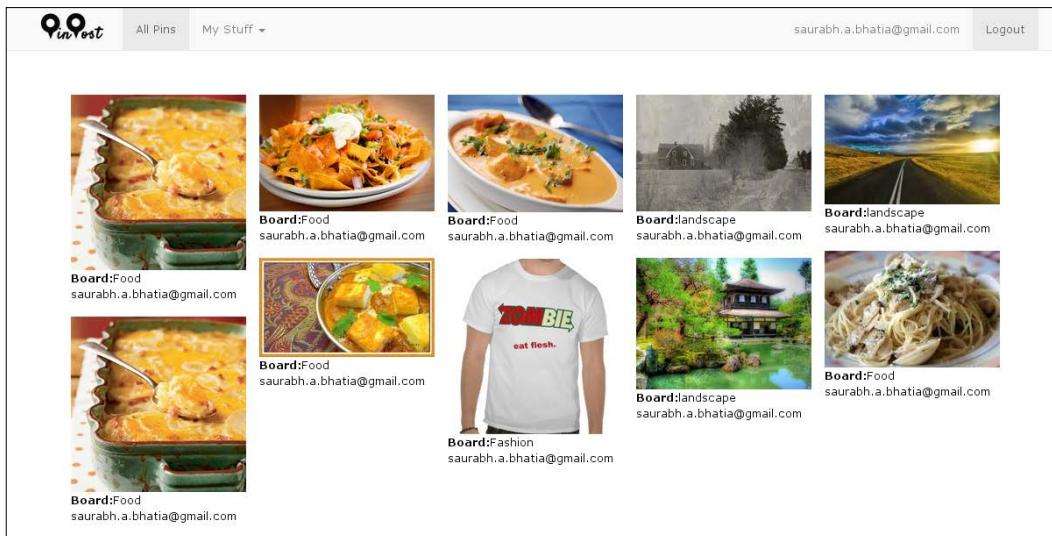
```
$handler.click(function() {
    var newHeight = $('img', this).height() + Math.round(Math.
random() * 300 + 30);
    $(this).css('height', newHeight+'px');
    // Update the layout.
    $handler.woomark();
});
```

As soon as we created the `td` element, our JavaScript automatically identified the element and generated it:

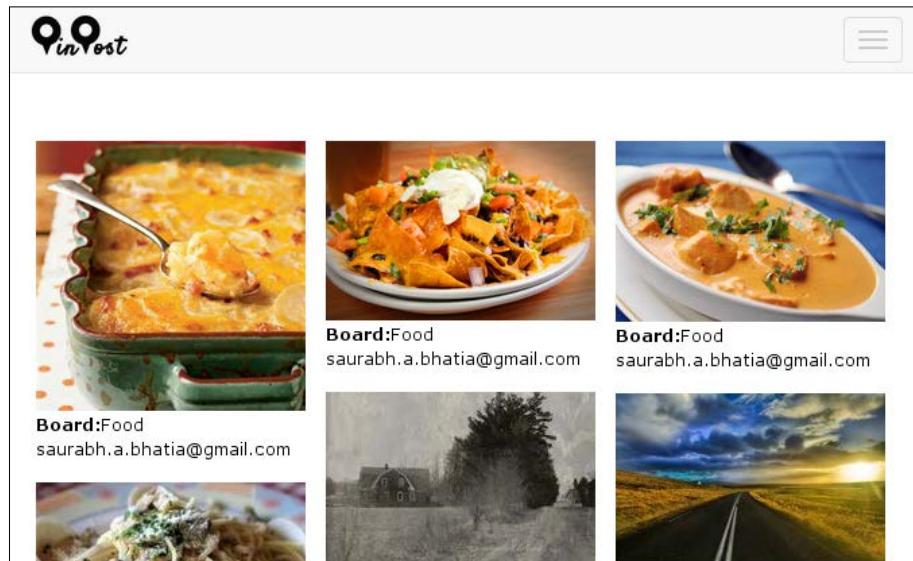
```
<td> <%=link_to(image_tag(pin.image.thumb.url), pin) %><p>
    <strong>Board:</strong><%= pin.board.title %><br/><%= pin.
    board.user.email %></p></td>
```

Creating an Online Social Pinboard

We have successfully generated the grid layout, which looks like that of Pinterest and is shown in the following screenshot:



The preceding layout is responsive too, so we will resize our browser as shown in the following screenshot and check it:



The layout being responsive also depends on Bootstrap as it contains media queries as a part of the CSS; however, Wookmark automates image resizing and grid size required for different window sizes. Hence, it is a completely responsive layout.

Adding a full-text search

Search is one of the most important functionalities today. Because sites are targeted at millions of users, there is a much larger volume of content. For a user to find what he or she is looking for, a full-text search is created. The idea of a search is to call the text, break it word by word, and match it with the key term supplied to it. We will use Apache Solr to create our search engine. In this section, we add a search option to our models using Sunspot, a Ruby-based library for Solr, indexing, and search methods in our Rails application.

Prepare for lift off

We will need to install Solr and Tomcat Solr before we start working with it. Solr relies on Java, so you need to have an updated version of OpenJDK before you proceed. Solr is generally a process bound with the sunspot gem and can be initiated using Rake. However, the Solr server depends on Tomcat and JDK, so they need to be installed before we start using Solr:

```
$ sudo apt-get install openjdk-6-jdk
```

Then install Tomcat and start the server:

```
sudo apt-get install solr-tomcat  
sudo service tomcat6 start
```

Engage thrusters

We will add a full-text search engine to our application:

1. Add sunspot and the supporting libraries to Gemfile and bundle install:

```
gem 'sunspot', :require => 'sunspot'  
gem 'sunspot_rails'  
gem 'sunspot_solr'
```

2. The main library Sunspot; sunspot_rails is specific to the interface with Rails applications and attaches it to the models. Sunspot Solr provides a Solr-related configuration interface.

3. Generate the configuration file:

```
:~/pinpost$ rails generate sunspot_rails:install  
      create config/sunspot.yml
```

4. The file looks like the following:

```
config/sunspot.yml  
production:  
  solr:  
    hostname: localhost  
    port: 8983  
    log_level: WARNING  
    # read_timeout: 2  
    # open_timeout: 0.5  
development:  
  solr:  
    hostname: localhost  
    port: 8982  
    log_level: INFO  
test:  
  solr:  
    hostname: localhost  
    port: 8981  
    log_level: WARNING
```

5. In case your Solr server is running on a different port, edit the port number in this file to match that. This will allow Solr to bind to that port and run on it.
6. We will load the Rake tasks manually. In Rails 4, the Rake tasks for Solr are not loaded by default. Thus, we will need to add them to our Rake file:

```
require 'sunspot/solr/tasks'
```

7. Start the Solr server using the Rake task:

```
:~/pinpost$ rake sunspot:solr:start  
java version "1.7.0_25"  
OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-  
lubuntu0.13.04.2)  
OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)  
Removing stale PID file at /home/user/pinpost/solr/pids/  
development/sunspot-solr-development.pid  
Successfully started Solr
```

8. Solr is now up and running on your system. Let's go ahead and add indexes on the fields we need to search.
9. Sunspot Solr accesses the database for full-text search through the models. We need to define these in our board and pin models:

```
models/board.rb
  searchable do
    text :title, :description
    integer :user_id
  end
models/pin.rb
  searchable do
    text :name, :image
    integer :board_id
  endOnce
```

10. The indices are set up; we will index the data to Solr.
11. Indexing is a Rake task of sunspot, so just run the following Rake command:

```
:~/pinpost$ rake sunspot:reindex
*Note: the reindex task will remove your current indexes and start
from scratch.

If you have a large dataset, reindexing can take a very long time,
possibly weeks.

This is not encouraged if you have anywhere near or over 1 million
rows.

Are you sure you want to drop your indexes and completely reindex?
(y/n)

y
#####
##### [88/88] [100.00%] [00:01] [00:00] [67.55/s]
```

12. We will now write our search methods in our model. We will then add a search method in order to search our indexed data.
13. In order to search the indexed data, Sunspot provides us with a search method. We will create a class method in our model to search through them. Our board model will look as follows:

```
app/models/board.rb
def self.search_board(search_key)
  @search = self.search do
    fulltext "#{search_key}"
  end
  @search.results
end
```

14. Our pin model will look like as shown in the following code snippet:

```
app/models/pin.rb
def self.search_pin(search_key)
  @search = self.search do
    fulltext "#{search_key}"
  end
  @search.results
end
```

15. We will call the `search` and `fulltext` methods as a `self` class method using a search term in the `search_key` variable. The `@search` object cannot be inspected. Hence, we call `@search.results` to output our results in the form of an object. Call the `search` method through the controller.

16. We will create a home controller in order to set up a home page for our application:

```
$ rails g controller home index
```

17. We will add our `search` method to the `home` controller as we want to create a site-wide search. First, check if the search term is blank. Also, we will check for a condition in which there are no results, and then display the message; else, we will add the results of the board and pin into one object:

```
app/controllers/home_controller.rb
def search
  if params[:search].blank?
    flash[:notice] = "Please Supply a Search term"
  If it is present, then search for board and pin
  else
    @board_results = Board.search_board(params[:search])
    @pin = Pin.search_pin(params[:search])
    if @board.nil? && @pin.nil?
      flash[:notice] = "No Results Found matching your query"
    else
      flash[:notice] = "Following are the search results"
      @search = @board + @pin
    end
  end
end
```

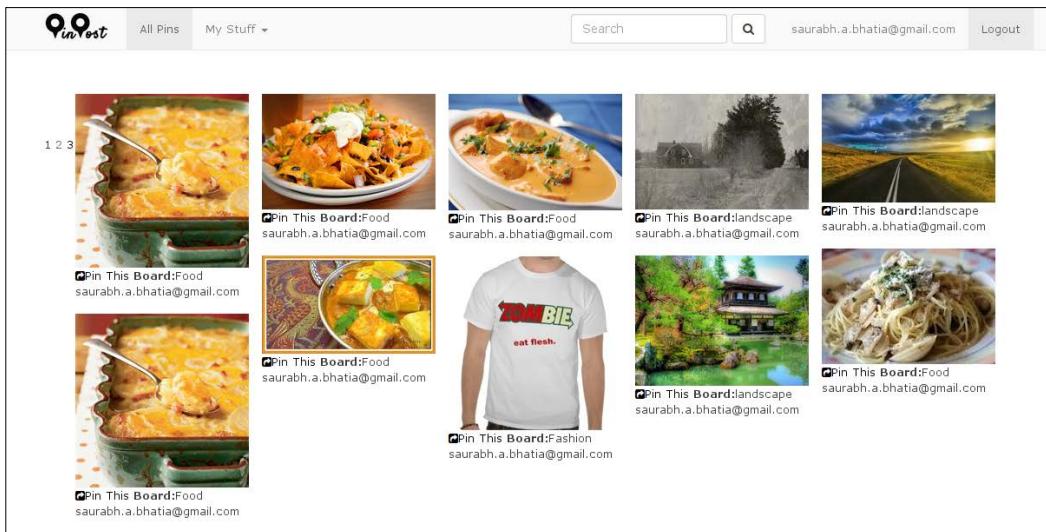
18. We now have the search results. In order to search from a form, we need a route. Our search method route will look like the following:

```
config/routes.rb
get :search, :to => 'home#search', :as => 'search'
```

19. However, to search from a form, we need to create a search form in layouts/application.html.erb.
20. This form will send the search term as params[:search], which will be passed to the controller method:

```
app/views/layouts/application.html.erb
<%= form_tag(search_path, :class=>"navbar-form navbar-left",
:method => :get) do%>
    <div class="form-group">
        <%=text_field_tag :search, params[:search], :class =>
'form-control', :placeholder => 'Search'%>
    </div>
    <button type="submit" class="btn btn-default">
        <i class="icon-search"></i>
    </button>
<%end%>
```

We have used the Font Awesome icon to create the search icon. We can now see a **Search** bar on the top of our page as shown in the following screenshot:



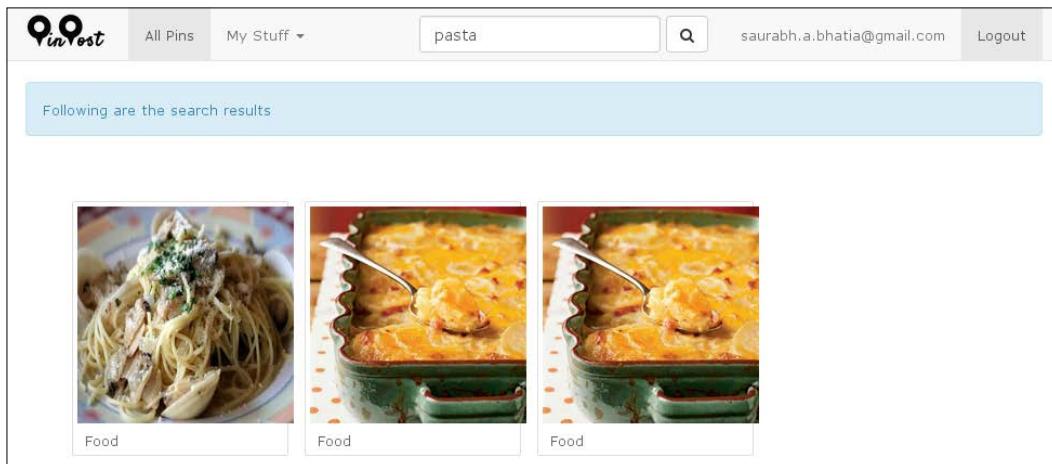
21. We will now display the search results by creating a search results page.
22. We can run the loop over our search object and identify if the class name is Board or Pin. In this way, we can differentiate between the different search results:

```
app/views/layouts/application.html.erb
<% @search.each do |s| %>
    <%if s.kind_of?(Board)%>
```

Creating an Online Social Pinboard

```
<li><p><%= s.title%><br/><%= s.description%><br/><%=  
s.user.email%></p></li>  
<%else if s.class.name == "Pin" %>  
<li><%=image_tag s.image.url%><p><%= s.board.title%></p></li>  
<%end%>  
<%end%>
```

This is how it looks after applying the Wookmark grid layout to the view:



Objective complete – mini debriefing

In this task, we added a full-text search engine to our application. We used Solr as our choice of search engine. Solr's website defines it as follows:



Solr™ is the popular, blazing, fast, open source, enterprise search platform in the Apache Lucene™ project. Its major features include powerful full-text search, hit highlighting, faceted search, near real-time indexing, dynamic clustering, database integration, rich document (for example, Word, PDF, and so on) handling, and geospatial search.

Although Solr has more dependencies than other counterparts (Sphinx and Elasticsearch), it is highly scalable and can handle complex queries with ease.

We first defined an index in the pin model:

```
searchable do  
  text :name, :image  
  integer :board_id  
end
```

In a search engine, indexing is a process to collect, parse, and store data such that it is matched and retrieved really quickly using the defined matching algorithms. A query on an index is much faster than a query on the database because an index remembers only a particular set of data, not the entire data set. An index is generally a data structure. Solr uses an inverted index data structure technique (a hash table or a binary tree) to implement the indexing function.

Once the index is defined, Sunspot generates an incremental index as soon as a new record is saved.

The `fulltext` method performs the search on the index created using the search term:

```
@search = self.search do
  fulltext "#{search_key}"
end
```

The results are returned as objects inside `@search`. In order to get the search results as a hash from the object, we called the following method:

```
@search.results
```

The terminal shows the Solr query once we try to search according to the search term:

```
Started GET "/search?utf8=%E2%9C%93&search=blade" for 127.0.0.1 at 2014-02-02
16:02:15 +0800
```

Processing by HomeController#search as HTML

```
Parameters: {"utf8"=>"✓", "search"=>"blade"}
```

```
SOLR Request (96.9ms) [ path=#<RSolr::Client:0x007f352c0d5008> parameters={data:
fq=type%3ABoard&q=blade&fl=%2A+score&qf=title_text+description_text&defType=di
smax&start=0&rows=30, method: post, params: {:wt=>:ruby}, query: wt=ruby, headers:
{"Content-Type"=>"application/x-www-form-urlencoded; charset=UTF-8"}, path: select,
uri: http://localhost:8982/solr/select?wt=ruby, open_timeout: , read_timeout: , retry_503:
, retry_after_limit: } ]
```

```
SOLR Request (27.0ms) [ path=#<RSolr::Client:0x007f352c0d5008> parameters={data: fq
=type%3APin&q=blade&fl=%2A+score&qf=name_text+image_text&defType=dismax&sta
rt=0&rows=30, method: post, params: {:wt=>:ruby}, query: wt=ruby, headers: {"Content-
Type"=>"application/x-www-form-urlencoded; charset=UTF-8"}, path: select, uri: http://
localhost:8982/solr/select?wt=ruby, open_timeout: , read_timeout: , retry_503: , retry_
after_limit: } ]
```

Rails 4.2 upgrade tip

In Rails 4.2, the require path, `active_support/core_ext/object/to_json`, is deprecated instead of `active_support/core_ext/object/json`.

Sunspot Rails gives the following deprecation warning:

DEPRECATION WARNING: You have required `active_support/core_ext/object/to_json`. This file will be removed in Rails 4.2. You should require `active_support/core_ext/object/json` instead. (called from <top (required)> at /home/rwub/rails4-book/book/6294OS_Chapter_03/project-3/config/application.rb:7)

```
sunspot_rails/lib/sunspot_rails.rb  
require 'active_support/core_ext/object/json'
```



Resharing the pins and creating modal boxes using jQuery

One of the most important features in our application is resharing. This is the most attractive feature businesswise and a USP of our application. If a user likes an image or pin, he or she would like to pin it up on their board. In this section, we will look at creating this functionality. This is the social aspect and also the business model. How do we check the most trending items? The number of times a pin has been shared can serve as a strong metric when suggesting a trending topic.

A user should be able to select the board on which you have put up the pin. These are the users' own boards. We can do this by creating a modal box with a list of users' boards in it.

Engage thrusters

In this task, we will add the functionality to repin the post:

1. Create a `pin_post` method in the pins controller:

- ❑ This method will call the pin and find it using the pin ID. Create a new pin and assign values to various attributes. We can save this pin once we build the complete object:

```
app/controllers/pins_controller.rb  
def pin_post  
  @current_pin = Pin.friendly.find(params[:id])  
  @pin = @current_pin.repin_post (params[:board_id])  
  respond_to do |format|
```

```
        if @pin.save
      format.js {render :layout => false}
        else
      format.js
        end
      end
    end
```

2. We will create a new pin in the pin model:

```
app/models/pin.rb
def repin_post(board_id)
  pin = Pin.new
  pin.name = self.name
  pin.board_id = board_id
  pin.image = self.image
  pin.save
end
```

3. Create a route to access this method from the controller.
4. We will pass the ID of the pin along with the route:

```
config/route.rb
post 'pin_post/:id', :to => 'pins#pin_post', :as => 'pin_post'
```

5. We will add a modal box using jQuery Facebox:

- Add the Facebox jQuery plugin using the `facebook-rails` gem:

```
Gemfile
gem 'facebook-rails'
```

- Add JavaScript to the `application.js` file:

```
app/assets/application.js
//= require jquery.facebox
```

- Also add the stylesheet to the `application.css` file

```
app/assets/application.css
*= require jquery.facebox
```

- Initiate facebox and ask it to bind to a tag with `rel="facebox"`:

```
app/views/pins/index.html.erb
<script>
jQuery(document).ready(function($) {
  $('a[rel*=facebox]').facebox()
})
</script>
```

6. In order to make this form reusable, create a partial called `pin_post.html.erb`:

```
app/views/pins/_pin_post.html.erb
<%= form_tag(pin_post_path(pin)) do%>
  <p>Select a board for Pinning</p>
  <% if current_user.boards%>
    <% current_user.boards.each do |b| %>
      <%= radio_button_tag :board_id, b.id %>
      <%= b.title %><br/>
    <%end%>
    <%= submit_tag 'Save', :class => "btn btn-primary"%>
  <%end%>
<%end%>
```

7. Create a link to the info box to display the boards.
8. We can call the `Pin This` link and use the share icon from the Font Awesome icon library:

```
app/views/pins/index.html.erb
<a href="#info" rel="facebox"><i class="icon-share-sign"></i>Pin This</a>
```

9. Add a hidden div and call the partial `pin_post.html.erb` in the div:

```
app/views/pins/index.html.erb
<div id="info" style="display:none;">
  <%= render partial: 'pin_post', locals: {pin: pin}%>
</div>
```

Objective complete – mini debriefing

We have now added the modal box and allowed users to repin a particular pin they like on their own boards. The idea behind a repin is that users will save the same pin on the board that's associated with them. So, in order to repin, we created a class method in the model. We called the current pin in an object using ID:

```
@current_pin = Pin.friendly.find(params[:id])
```

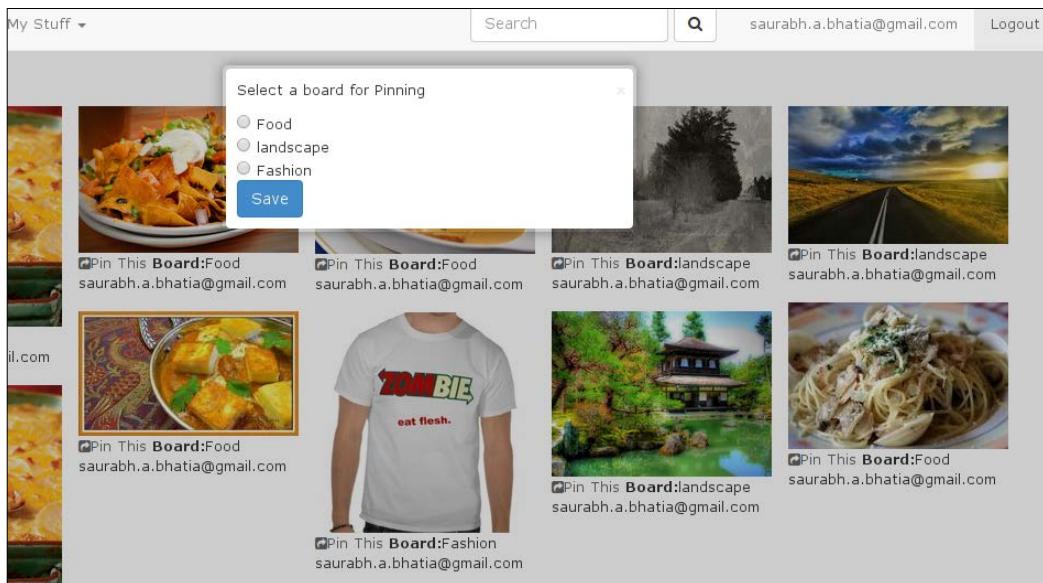
Then, we called a method to generate a new pin based on the information of the existing pin:

```
def repin_post(board_id)
  pin = Pin.new
  pin.name = self.name
  pin.board_id = board_id
  pin.image = self.image
  pin.save
end
```

We displayed the boards in a modal box. We used the jQuery Facebox plugin as the modal box. We used Facebox (<https://github.com/defunkt/facebook>) to load a partial with a form that contains the list of boards selectable using radio buttons. The `form_tag` binds to the controller action via the route. The `remote=>true` enables the AJAX form submission. Also, `radio_button_tag` generates the radio button to select the board value:

```
<%= form_tag(pin_post_path(pin), :remote=>true) do %>
  <p>Select a board for Pinning</p>
  <% Board.my_board(current_user).each do |b| %>
    <%= radio_button_tag :board_id, b.id %>
    <%= b.title %><br/>
  <%end%>
  <%= submit_tag 'Save', :class => "btn btn-primary" %>
<%end%>
```

The modal box for resharing a pin looks like what is shown in the following screenshot:



Enabling the application to send a mail

Mailers are the oldest way of marketing and still prove to be one of the most effective ways to reach out to users. As a part of the user engagement model, we can create a weekly mailer with a list of recent pins. This will keep the user updated with the latest information posted on our website and enhance users' engagement.

Engage thrusters

In the coming steps, we will create a mailer service for our application. To do so, we need to set up Action Mailer and use the Google apps e-mail service to send the mails via our application:

1. Add the following lines inside your `development.rb/production.rb` file:

```
config/environments/development.rb
config.action_mailer.smtp_settings = {
  :enable_starttls_auto => true,
  :address => "smtp.gmail.com",
  :port => '587',
  :domain => "smtp.gmail.com",
  :authentication => "plain",
  :user_name => "foobar@pinpost.com",
  :password => "myawesomewp" }
```

2. We will have to make sure that we do not commit the credentials of our mailer system in our version control; you can use dummy credentials. We can also avoid sending out mail in the development mode by using MailCatcher. We can install it using the `gem` command first:

```
$ gem install mailcatcher
Fetching: skinny-0.2.3.gem (100%)
Fetching: mailcatcher-0.5.12.gem (100%)
Successfully installed skinny-0.2.3
Successfully installed mailcatcher-0.5.12
2 gems installed
```

Then, start the MailCatcher service using the `mailcatcher` command:

```
$ mailcatcher
Starting MailCatcher
==> smtp://127.0.0.1:1025
==> http://127.0.0.1:1080
*** MailCatcher runs as a daemon by default. Go to the web
interface to quit.
```

3. Now we will edit our mailer settings in our `environments/development.rb` file:

```
config/environments/development.rb
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = { :address => "localhost",
:port => 1025 }
```

4. We will then generate a mailer called newsletter:

```
:~/pinpost$ rails g mailer newsletter
      create  app/mailers/newsletter.rb
      invoke  erb
      create    app/views/newsletter
      invoke  test_unit
      create    test/mailers/newsletter_test.rb
```

5. We will add a mailer method called letter in order to send the e-mail.

6. Then, we will pass the user e-mail and pin as the argument and pick up the e-mail from it:

```
app/mailers/newsletter.rb
class Newsletter < ActionMailer::Base
  default from: "noreply@pinpost.com"
  def letter(user, pin)
    @user = user
    @pins = pin
    mail(:to => @user.email, :subject => "Latest Pins from Our
Users")
  end
end
```

7. Next up, we will create a controller method to retrieve these objects and pass them to the mailer method.

8. We will create a pins_newsletter method, calling the last five pins into an object and looping them over all users:

```
app/models/pin.rb
def self.send_newsletter
  @user = User.all
  @user.each do |u|
    @pins = self.all(:limit => 5)
    Newsletter.letter(u, @pins).deliver
  end
end
```

9. We will need to add e-mail views to the newsletter folder located under views.

10. The newsletter folder was created when we generated the mailer earlier. This folder will hold the views for the e-mail, that is, how the e-mail will look. In our case, we will call the last five pins and directly link to them via e-mail and call the letter.text.erb file:

```
app/VIEWS/newsletter/letter.text.erb
<h3>Our Latest Pins</h3>
<% @pins.each do |p| %>
  <%=link_to p.name, p %>
<%end%>
```

11. Now that we are ready with our methods, we need to create a Rake task as we want this functionality to run in the backend.
12. Create a newsletter.rake file in lib/tasks.
13. This will basically invoke a new method with an instance called pins_newsletter and send an e-mail to all the users:

```
lib/tasks/newsletter.rake
namespace :newsletter do
  desc "Send Newsletter"
  task :send => :environment do
    Pin.send_newsletter
  end
end
```

We will bind the preceding Rake task to cron using the whenever gem.

14. Add the whenever gem to Gemfile and bundle:

```
gem 'whenever', :require => false
```
15. Generate the configuration file called schedule.rb:

```
:~/pinpost$ wheneverize .
[add] writing `./config/schedule.rb'
[done] wheneverized!
```
16. Configure the Rake task to run every seven days in order to send a weekly mail:

```
config/schedule.rb
every 7.days do
  rake "newsletter:send"
end
```

17. Update the crontab file:

```
:~/pinpost$ whenever --update-crontab store  
[write] crontab file updated
```

18. Check if the crontab file has been updated by listing the cron jobs:

```
:~/pinpost$ crontab -l  
# Begin Whenever generated tasks for: store  
0 0 1,8,15,22 * * /bin/bash -l -c 'cd /home/rwub/rails4-book/  
book/6294OS_Chapter_03/project-3 && bin/rails runner -e production  
'\\'Send Newsletter Email'\\'  
0 0 1,8,15,22 * * /bin/bash -l -c 'cd /home/rwub/rails4-book/  
book/6294OS_Chapter_03/project-3 && RAILS_ENV=production bundle  
exec rake newsletter:send --silent'
```

Objective complete – mini debriefing

In this task, we saw how to create a mailer using Action Mailer and bind it to cron in order to send weekly e-mails.

We first created a mailer in our application. The mailer then creates a class under the `mailers` folder. We defined the `default from` e-mail in our mailer class. Next, as a part of the e-mail, we defined a method with `user` and `pins` as attributes. Finally, we added the subject of our e-mail and sent it to all the users in the system:

```
app/mailers/newsletter.rb  
class Newsletter < ActionMailer::Base  
  default from: "noreply@pinpost.com"  
  def letter(user, pin)  
    @user = user  
    @pins = pin  
    mail(:to => @user.email, :subject => "Latest Pins from Our Users")  
  end  
end
```

Then we defined the text for the e-mail to be sent out. In that, we displayed the list of the last five pins that have been created:

```
app/views/newsletter/letter.text.erb  
<h3>Our Latest Pins</h3>  
<% @pins.each do |p| %>  
  <%= link_to p.name, p %>  
<% end %>
```

In order to fetch this information and fire the method, we created a class method in the `Pin` controller. This method will fetch all the users and the last five pins. We will pass the `user` object and `pins` object to the `Newsletter` mailer class. Also, `Model.all` is deprecated in Rails 4. A direct replacement for the `Model.all` call is `Model.to_a`. In order to limit the number of records to be selected in the query, we will first have to pass the `order` argument and then the `limit` argument:

```
def self.send_newsletter
  @user = User.all.to_a
  @user.each do |u|
    @pins = self.order('id ASC').limit(5)
    Newsletter.letter(u, @pins).deliver
  end
end
```

We fired this using a Rake task that directly calls the `send_newsletter` method in the `Pin` model:

```
namespace :newsletter do
  desc "Send Newsletter"
  task :send => :environment do
    Pin.send_newsletter
  end
end
```

In order to send e-mails on a periodic basis, we added the `cron` jobs to our application and used the `whenever` gem for it. The `whenever` gem uses a file called `schedule.rb` to define that our task will run every seven days:

```
Config/schedule.rb
every 7.days do
  rake "newsletter:send"
end
```

The `whenever` gem edits the Linux `cron` jobs in order to send e-mails from time to time. The e-mails in our terminal look like the following:

```
Sent mail to myawesomeuser@gmail.com (15.8ms)
Date: Mon, 02 Sep 2013 07:11:29 +0800
From: noreply@pinpost.com
To: myawesomeuser@gmail.com
Message-ID: <5223c9a1d7189_ed53fb18a55acb44652d@rwub.mail>
```

Subject: Latest Pins from Our Users

Mime-Version: 1.0

Content-Type: text/plain;

charset=UTF-8

Content-Transfer-Encoding: 7bit

<h3>Our Latest Pins</h3>

Zombie Tshirt

Kyoto Ginkakuju Temple

Spaghetti Cheese

Pasta Au Gratin

Long road

We can also browse to localhost :1080 to see the sent e-mails in our **MailCatcher** web console. The screenshot shows the output of the previous code:

The screenshot shows the MailCatcher web interface. At the top, there is a search bar labeled "Search messages..." and buttons for "Clear" and "Qu". Below the header, there is a table with columns: "From", "To", "Subject", and "Received". Two rows of data are visible:

From	To	Subject	Received
<noreply@pinpost.com>	<safewlabs@gmail.com>	Latest Pins from Our Users	Sunday, 2 Feb 2014 0:42:46 AM
<noreply@pinpost.com>	<saurabh.a.bhatia@gmail.com>	Latest Pins from Our Users	Sunday, 2 Feb 2014 0:42:46 AM

Below the table, a message preview is shown. It includes the "Received" timestamp, the "From" address (<noreply@pinpost.com>), the "To" address (<saurabh.a.bhatia@gmail.com>), and the "Subject" ("Latest Pins from Our Users"). There are three tabs at the bottom of the preview area: "Plain Text", "Source", and "Analysis". The "Plain Text" tab is selected. The content of the email is displayed as:

```
<h3>Our Latest Pins</h3>
<a href="/pins/i-love-to-paneer">Au Gratin Pasta</a>
<a href="/pins/i-love-to-paneer-e75297a5-ee3c-4a65-8e0e-2ad792ebe2c3">I love to Paneer</a>
<a href="/pins/i-love-to-paneer-e9444778-c677-452e-8af4-7a8886831109">I love to Paneer</a>
<a href="/pins/i-love-to-paneer-dc084a28-f595-4e41-9f60-2c8742a55b24">I love to Paneer</a>
<a href="/pins/i-love-to-paneer-8de6487b-83e2-461b-a8b8-c04ac5aafefb">Long road</a>
```

We can set weekly mail as a background job as well using **Sidekiq** or **Resque**. We will look at Sidekiq in later projects. It is a much advanced version of creating background jobs and job queues and is used in cases where there are several asynchronous jobs to be run in the background.

Securing an application from cross-site scripting or XSS

The Internet comes with its share of security concerns. There are several types of attacks you will have to avoid while working with your Rails application: session hacking, cookie stealing, SQL injections, and cross-site scripting. In this section, we will only look at cross-site scripting and how to avoid it in our application.

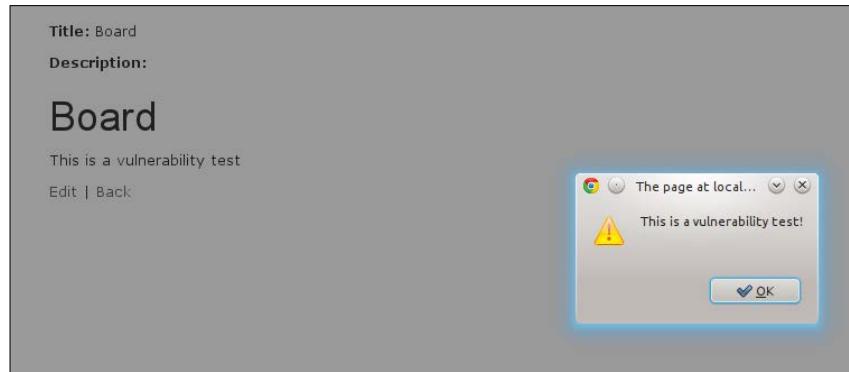
Engage thrusters

The following steps will give us some security tips:

1. Check for vulnerability by adding a simple alert box in your text area field.
2. Create a new board and add the following code in your description area:

```
<h1>Board</h1>
<p>This is a vulnerability test</p>
<p><script>alert('This is a vulnerability test!');</script></p>
```

3. If your application gives out an alert whenever the page loads, your site is vulnerable to cross-site scripting:



4. This can occur even if we apply the `html_safe` filters and allow HTML to be passed as part of the text. By default, Rails sanitizes all HTML to text and uses the latest HTML5 standards in order to do so.

5. We need to escape HTML in order to stop the JavaScript execution.
6. We will use the Rails HTML escape helper in order to escape the HTML in textboxes and prevent the execution of JavaScript in our text area. This will ensure the security of our application from JavaScript attacks:

```
<%= (@board.description) %>%>
```

Objective complete – mini debriefing

We have sanitized the HTML so that JavaScript is not inserted into our textboxes and executed every time the page is loaded. Cross-site scripting is a very serious issue. It could lead to concerns such as session and cookie stealing. A malicious user can enter such a JavaScript in our database and steal session information every time the page is loaded.

Rails has several forms of security built into the framework; let's be smart enough and use them.

Mission accomplished

In this project, we have created a simple social sharing website. We discussed creating pins and boards and resharing the pins. We looked at various jQuery libraries—Infinite scroll, Facebox, and Wookmark—and how to quickly use them to our advantages. We also used Solr to create a full-text search engine for our website.

We created a weekly mailer to increase our user engagement and used the cron job to make it a periodic task that runs in the background. Last but not least, we looked at potential security vulnerabilities and a simple way to fix these issues.

Hotshot challenges

Great! We have achieved a lot at the end of this project. Give yourself a pat on your back. Now it's time to take these concepts ahead and try out new things with what we've seen in this project:

- ▶ Use Amazon S3 instead of the filesystem to upload files
- ▶ Count the number of repins for each pin
- ▶ Add facets
- ▶ Write integration tests for the search option using minitest
- ▶ Create a mailer with the five most shared pins

Project 4

Creating a Restaurant Menu Builder

Tablets and smartphones are becoming cheaper by the day. They have better screens, faster processors, and better graphics. Hence, it makes perfect sense to port the restaurant menus to tablets and smartphones. It will save a lot of issues related to print and design and also save a lot of cost by making the menus easy to update. It will also make the process of ordering much faster and less prone to human errors. In this project, we will build a SaaS-based product to create restaurant menus online.

Mission briefing

In this project, we will create a SaaS-based software to create restaurant menus. Users can sign up and will have their own subdomain and area; they will also have plans to select from. Along with this, they can also create products and menus and assign them to a restaurant.

While building this project, we will take a look at concepts such as concerns, subdomains, creating plans, and managing a SaaS-based product. We will also see various ways to add roles to our application users, multitenancy in applications, and import and export data in various formats. Using these techniques, we can end up refactoring our code.

Our SaaS application's home page looks like the following at the end of our project:

The screenshot shows a web application interface for a restaurant menu builder. At the top, there is a dark header bar with the logo 'Go-table' on the left, a 'Home' button in the center, and 'Login' and 'SignUp' buttons on the right. Below the header is a large image of a restaurant interior with people dining. Overlaid on the image is the text 'Build and Manage Menus With Ease!' and 'Image Source : Taki Lab'. A blue button labeled 'Sign up today' is visible. Below the image, there is a section titled 'Plans' with three plan options: 'Small - 10\$ / Month', 'Medium - 30\$ / Month', and 'Large - 60\$ / Month'. Each plan table lists features such as the number of restaurants, tables, menu items, and storage space, along with a 'Sign Up' button.

Small - 10\$ / Month	Medium - 30\$ / Month	Large - 60\$ / Month
1 Restaurant	5 Restaurants	10 Restaurants
10 tables / Restaurant	50 tables / Restaurant	100 tables / Restaurant
20 Menu Items / Restaurant	50 Menu Items / Restaurant	50 Menu Items / Restaurant
5 GB storage	10 GB storage	50 GB storage
Sign Up	Sign Up	Sign Up

Why is it awesome?

SaaS-based applications (such as <http://basecamp.com/>) are one of the most popular business models these days. As a lot of offline businesses are moving towards cloud, we will build a SaaS-based restaurant management application. We will create a multiple plan-based system where users can pay on a monthly basis after a free trial for a limited set of resources. The system will also allow data to be imported and exported in the CSV format.

At the end of this project, we will be able to build a framework for a SaaS-based application.

Your Hotshot objectives

While building this application, we will have to go through the following tasks:

- ▶ Creating organizations with signup
- ▶ Creating restaurants, menus, and items
- ▶ Creating user roles

- ▶ Creating plans
- ▶ Creating subdomains
- ▶ Adding multitenancy and reusable methods
- ▶ Creating a monthly payment model, adding a free trial plan, and monthly billing
- ▶ Exporting data to a CSV format

Mission checklist

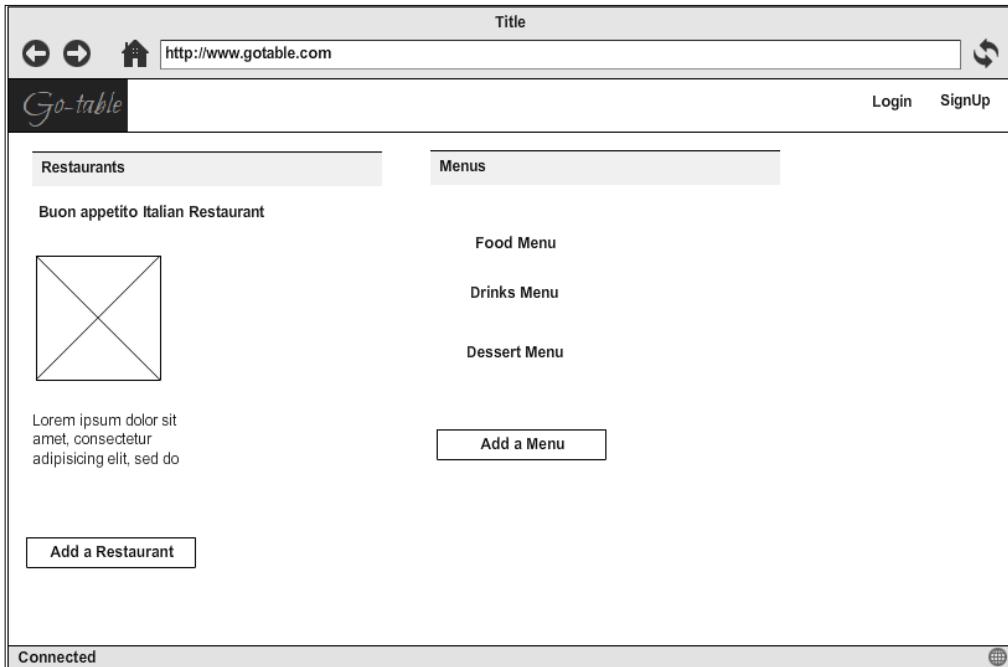
We need the following installed on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MySQL 6
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Sublime Text
- ▶ Devise
- ▶ Git
- ▶ A tool for mockups
- ▶ jQuery
- ▶ ImageMagick and RMagick

Creating organizations with sign up

Every user who creates a **SignUp** for our application will need an organization. This is because a business is not run in isolation, and there will be different stakeholders in the system; the service staff, chefs, and managers will all need access to the system. Now that we have defined the roles of different types of users of the system, we will bring them together on one level of abstraction called organization. In this task, we will build a wizard to set up an organization as a part of the **SignUp** process. We will use the `wicked` gem to create a wizard in our application.

The following is our standard protocol of mocking up our page—we will first use our wireframing tool to create a mockup of our home screen. Our mockup for the home screen looks like the following screenshot:



Prepare for lift off

Before we begin this task, we need to generate a blank application. Then, we need to create models for organization and user. We can create a `users` model using the `devise` gem. In order to create organizations, we first create the model:

```
gotable$ rails g model name:string description:text
```

Then, we will create a relationship between `users` and `organizations`:

```
app/models/organization.rb
  has_and_belongs_to_many :users
app/models/user.rb
  has_and_belongs_to_many :organizations
```

In order to store the association data, we will create a table:

```
gotable$ rails g migration organizations_users organization_id:integer
          user_id:integer
```

Engage thrusters

To create an organization, we will perform the following steps:

1. We will first add the wicked gem to the Rails Gemfile and run bundle install:

Gemfile

```
gem 'wicked'  
:~/gotable$ bundle install
```

2. We will then generate a controller in order to define these steps:

```
:~/gotable$ rails g controller setup_organization  
create app/controllers/setup_organization_controller.rb  
invoke erb  
create app/views/setup_organization
```

3. We will start by defining the steps inside `setup_organization_controller.rb` and include the wicked wizard module in it to autoload the wicked module as soon as this controller is called:

```
app/controllers/setup_organization_controller.rb  
class SetupOrganizationController < ApplicationController  
  include Wicked::Wizard  
  steps :organization_setup  
end
```

4. The first step of our wizard is signup. Hence, we will modify the signup method so that it redirects to the step defined in wicked once it is executed. We will edit `devise/registrations_controller` to suit our needs:

```
app/views/devise/registrations_controller.rb  
def after_sign_up_path_for(resource)  
  session[:plan_id] = params[:plan_id]  
  setup_organization_path(:organization_setup)  
end
```

5. The wicked gem uses the show and update actions in order to perform most of the tasks. The show action is used in order to initiate the step and render the page. Then, the update action is used in order to send the variables to the respective model:

```
app/controllers/setup_organization_controller.rb  
def show  
  @user = current_user  
  case step  
  when :organization_setup
```

```
    @organization = Organization.new
  end
  render_wizard
end
```

6. We need a form to submit these values to the database, and the form resource needs to point to the wizard path.
7. The form for the organization setup submits to the wizard path as a `put` method instead of a `post` method:

```
app/views/organizations/_wizard.html.erb
<%= form_for(@organization, :url => wizard_pathh, :method =>
:put) do |f| %>
  <% if @organization.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@organization.errors.count, "error") %>
prohibited this organization from being saved:</h2>
      <ul>
        <% @organization.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <div class="form-group">
    <%= f.label 'Organization Name' %>
    <%= f.text_field :name, :class=>"form-control", :placeholder
=> "Organization Name" %>
  </div>
  <div class="form-group">
    <%= f.label :description %>
    <%= f.text_area :description, :class=>"form-control",
:rows=>"3", :placeholder => "Description" %>
  </div>
  <%= f.submit 'Create', :class=>"btn btn-default" %>
<% end %>
```

8. However, once the form is filled and submitted, we need to send the params to the `organization` table. We will do so via the `update` action of `wicked` in our controller.

9. In our case, we have a **has_and_belong_to_many (HABTM)** relationship between organization and users. Hence, we will create @organization.users as a hash:

```
app/controllers/setup_organizations_controller.rb
def update
  @user = current_user
  @organization = Organization.new(organization_params)
  @organization.users << @user
  render_wizard @organization
end
```

10. However, we need to whitelist organization_params in our controller to access them in the method:

```
app/controllers/setup_organizations_controller.rb
private
  # Never trust parameters from the scary internet, only allow
  # the white list through.
  def organization_params
    params.require(:organization).permit(:name, :description,
    :plan_id, { :user_ids => [] })
  end
```

11. Note that we have added user_ids as an array because it points to the join_table representing has_and_belongs_to_many relationship.

12. We need to end the wizard once the steps are completed:

```
app/controllers/setup_organizations_controller.rb
private
def redirect_to_finish_wizard
  redirect_to dashboard_path, notice: "Thank you for signing up.
  You can now build beautiful menus"
end
```

13. Lastly, we need a route to tie this all up. The route will build the resource according to the controller name and will be assigned to the wizard_path in our view:

```
config/routes.rb
resources :setup_organization
```

14. The following screenshot shows what **Step 1** in the form looks like now:

The screenshot shows a web application interface titled "Setup a user - Step 1". At the top left is the logo "Go-table". To its right is a "Home" link. Below the title, there are four input fields labeled "Domain name", "Email", "Password", and "Password confirmation". Each field has a placeholder text: "Enter your domain name", "Email", "Password", and "Password Confirmation" respectively. Below these fields is a "Sign up" button. Underneath the button, there are two links: "Sign in" and "Forgot your password?".

Objective complete – mini debriefing

In the previous task, we looked at how to create a wizard in our application. We saw the concept of `has_and_belongs_to_many` in the context of Rails 4 in this task. HABTM is a special case of relationships in which both models have a many-to-many relationship with each other. In our `users` and `organizations` models, we wrote the following:

```
has_and_belongs_to_many :organizations  
has_and_belongs_to_many :users
```

We created a table with `organization_id` and `user_id`. This table will store the data for the organizers and associated users. We chose to do it this way because we did not want to do anything further with the association. In case we want to do more with the association of the two models, we can create a third model and define the relationship in the following way:

```
has_many :organizations, :through => :members  
has_many :users, :through => :members
```

We will have to create a separate `members` model with the following associations:

```
belongs_to :organization  
belongs_to :user
```

Coming back to our code, in order to assign a user to organizations, we need to pass it as an array:

```
@organization.users << @user
```

In our organization controller, we added a `user_ids` array to our `params` user so that multiple users can be associated to an organization:

```
params.require(:organization).permit(:name, :description,  
:plan_id, { :user_ids => [] })
```

A wizard significantly improves the engagement and user experience in a website because it decreases the number of fields in a form. However, too many steps can also lead to the same problem. Hence, a balance needs to be attained between the steps and fields that will go as a part of our wizard. The `wicked` gem is quite a comprehensive solution as far as wizards are concerned, as you can also build a single object across different steps.

In this task, we included the `wicked` gem and defined the steps for our wizard. The `wicked` gem completely relies on the `ActionController` module of Rails. We created a new controller called `setup_organizations` to define the steps required for the wizard. The wizard identifies the controller using a module inclusion:

```
include Wicked::Wizard
```

When we define steps, they are called one by one inside the `show` action. This is because `wicked` binds itself to the object ID in order to create the flow of steps. In order to access the ID, we need to prefix it with the object name; for example, `plan_id`. The `wizard_path` picks up the path of the current step in the wizard. Once the `include` method is defined in the controller, the controller and step path are substituted in `wizard_path`. The same procedure follows for the subsequent steps.

As `wicked` relies on the object ID, it uses `show` and `update` methods to generate the steps. Each step will have a view of the same name. The `show` action initializes and starts the wizard. The `update` action updates the `params` at each step. At the end of the wizard, runs the `update` query for the record.

In order to end the wizard, we used a custom private method called `redirect_to_finish_wizard`. This method in `wicked` allows us to redirect to the page we want to:

```
def redirect_to_finish_wizard  
end
```

Finally, we added a route to access all the methods inside the `setup_organization` controller.

Step 2 in the form in the wizard looks like the following screenshot:

The screenshot shows a web-based application interface for creating a new organization. The title bar says "Go-table". The main content area is titled "Setup Your Organization - Step 2". It has two input fields: one for "Organization name" containing "Sally Pizza and Pasta" and another for "Description" containing "A Pizza Shop with a difference.". At the bottom is a "Create" button.

Creating restaurants, menus, and items

The organization in our case is a company that owns the restaurants. There can be one or more restaurants inside an organization. These restaurants will have different menus; for example, a menu for dessert, a menu for the main course, and a menu for the drinks. These menus are going to have many items. We will add these items by creating nesting between menus and items.

Prepare for lift off

We first have to create models for restaurants, menus, and items with their respective attributes:

```
rails g scaffold restaurant name:string description:text slug:string
rails g scaffold menu title:string description:text
rails g model item name:string description:text price:float
```

Engage thrusters

1. We will add restaurants, menus, and menu items to our application. For the sake of convenience, we will generate a scaffold of the `restaurant` model. Please be sure to delete the `scaffold.css.scss` file because it conflicts with the existing CSS file in the system and tends to override it in places. The following code shows what the model for a restaurant looks like:

```
app/models/restaurant.rb
class Restaurant < ActiveRecord::Base
```

```
extend FriendlyId
friendly_id :name, use: :slugged
has_many :menus
belongs_to :organization
validates :name, presence: true
end

2. We will pass organization_id in the create method in the restaurant controller. The organization ID is our way to lock down all restaurants, menus, and items according to a single organization:

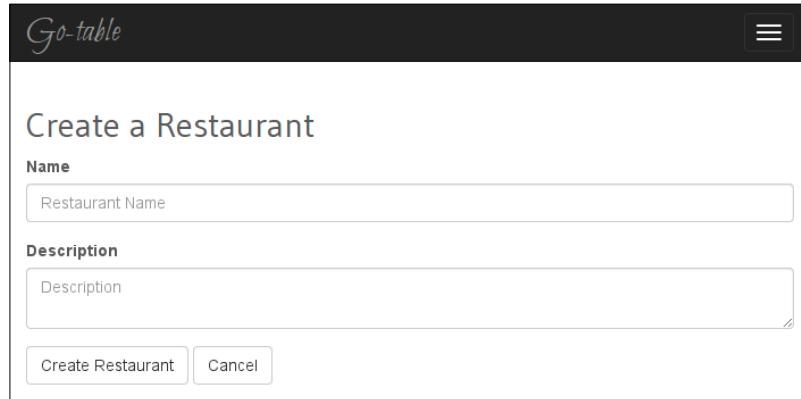
controllers/restaurants_controller.rb
def create
  @restaurant = Restaurant.new(restaurant_params)
  @restaurant.organization_id = current_user.organizations.
first.id
  respond_to do |format|
    if @restaurant.save
      format.html { redirect_to @restaurant, notice: 'Restaurant was successfully created.' }
      format.json { render action: 'show', status: :created,
location: @restaurant }
    else
      format.html { render action: 'new' }
      format.json { render json: @restaurant.errors, status:
:unprocessable_entity }
    end
  end
end

def restaurant_params
  params.require(:restaurant).permit(:name, :description,
:organization_id)
end

app/views/_form_errors.html.erb
<% if @restaurant.errors.any? %>
<div id="error_explanation">
  <h2><%= pluralize(@restaurant.errors.count, "error") %>
prohibited this restaurant from being saved:</h2>
  <ul>
    <% @restaurant.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
  </ul>
</div>
<% end %>
```

```
views/restaurants/_form.html.erb
<%= form_for(@restaurant) do |f| %>
  <%= render 'form_errors'%>
  <div class="form-group">
    <%= f.label :name %><br>
    <%= f.text_field :name, :class=>"form-control", :placeholder
=> "Restaurant Name" %>
  </div>
  <div class="form-group">
    <%= f.label :description %><br>
    <%= f.text_area :description, :class=>"form-control",
:placeholder => "Description" %>
  </div>
  <div class="actions">
    <%= f.submit :class=>"btn btn-default" %> <%= link_to
'Cancel', restaurants_path, :class=>"btn btn-default" %>
  </div>
<% end %>
```

At the end of this iteration, our restaurant view looks like the following screenshot:



3. The `Menu` class is similar to the `restaurant` class. The main difference here is that we will add support for nesting with items. We will add `accepts_nested_attributes_for :items` so that we can access parameters of items within the menu model:

```
app/models/menu.rb
class Menu < ActiveRecord::Base
  belongs_to :restaurant
  has_many :items
  accepts_nested_attributes_for :items
end
```

4. On the restaurants show page, we need to add `restaurant_id` as a parameter so that it is passed as a parameter to find the menus related to a particular restaurant:

```
controllers/restaurants_controller.rb
def show
  @menus = Menu.where(:restaurant_id => @restaurant.id)
end
views/restaurants/show.html.erb
<h3>Menus</h3>
<% @menus.each do |m| %>
  <%= link_to m.title, menu_path(m) %>
<%end%>
<%= link_to "Add a Menu", new_menu_path(:restaurant_id => @
restaurant.id), :class=>"btn btn-default" %>
```

5. Items do not have a controller and view separately. They will reside as a part of menu in our application. So, the attributes of items need to be whitelisted inside our `menus_controller` class:

```
app/controllers/menus_controller.rb
# Never trust parameters from the scary internet, only allow the
white list through.
def menu_params
  params.require(:menu).permit(:title, :description,
:restaurant_id,
:items_attributes => [:id, :name, :description,
:price,
:_destroy])
)
end
```

6. Associate the `item` model with the menu:

```
app/model/item.rb
class Item < ActiveRecord::Base
  belongs_to :menu
end
```

7. We now need to build the views. At this point, we need to add the `nested_form` gem to our application:

```
Gemfile
gem "nested_form"
```

8. Add `jquery_nested_form` script to `assets/application.js`:
`app/assets/application.js`
`//= require jquery_nested_form`
9. We will first convert our menu form to a nested form by adding `nested_form_for`:
`app/views/menus/_form.html.erb`

```
<%= nested_form_for(@menu) do |f| %>
  <% if @menu.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@menu.errors.count, "error") %> prohibited
      this menu from being saved:</h2>
      <ul>
        <% @menu.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <div class="form-group">
    <%= f.label :title %><br>
    <%= f.text_field :title, :class=>"form-control", :placeholder
=> "Title" %>
  </div>
  <div class="form-group">
    <%= f.label :description %><br>
    <%= f.text_area :description, :class=>"form-control",
:placeholder => "Description" %>
  </div>
  <%= f.hidden_field :restaurant_id, :value => params[:restaurant_
id] %>
  <div class="actions">
    <%= f.submit 'Create Menu', :class=>"btn btn-default" %>
  </div>
<% end %>
```

10. In order to add items, we will add `fields_for` to the nested form:

```
app/views/menus/_form.html.erb
<%= f.fields_for :items do |i| %>
  <div class="form-group">
    <%= i.label :name %><br>
    <%= i.text_field :name, :class=>"form-control", :placeholder
=> "Name" %>
```

```
</div>
<div class="form-group">
<%= i.label :price %><br>
<%= i.text_field :price, :class=>"form-control", :placeholder => "Price" %>
</div>
<div class="form-group">
<%= i.label :description %><br>
<%= i.text_area :description, :class=>"form-control", :placeholder => "Description" %>
</div>
<%= i.link_to_remove "Remove this item", :class=>"btn btn-default" %>
<% end %>
<p><%= f.link_to_add "Add an Item", :items, :class=>"btn btn-default" %></p>
```

Objective complete – mini debriefing

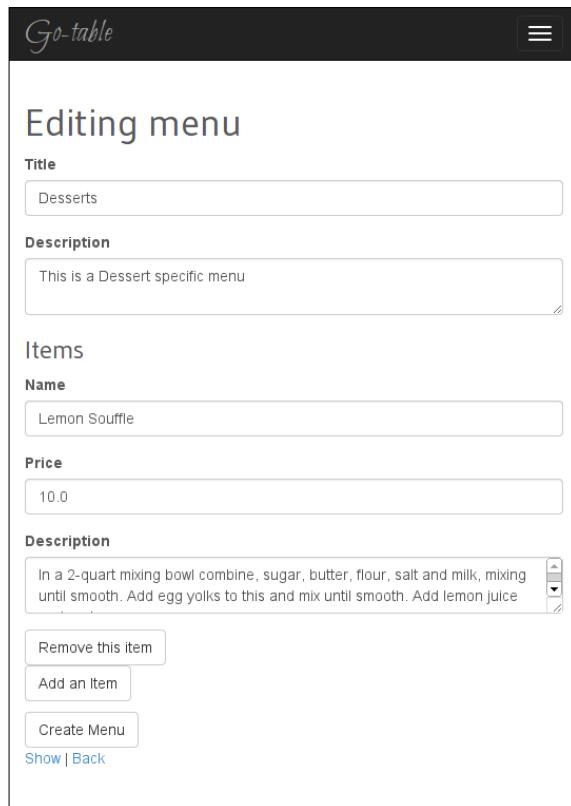
In this task, we first created models for our restaurant, menu, and item.

Using a nested form is a very useful technique and helps reduce a lot of unnecessary code. It is helpful also because it keeps the application structure easy to understand and the flow logical. We used the `nested_form` gem to create a form for items within our menu form. The gem depends on jQuery and allows the developer to create multiple nested forms inside a model. In order to make it work with strong parameters, we added the `:_destroy` method to our parameters. This will allow the deletion of the nested model records. In order to make our form recognize methods from the gem, we have to modify `form_for` to `nested_form_for`:

```
<%= nested_form_for(@menu) do |f| %>
```

The gem adds extra form helpers (such as `nested_form_for`) on top of Rails in order to generate the nested form. Some other form helper methods are `link_to_add` or `link_to_remove` that add or remove the tasks. The gem also creates an interface to jQuery in order to add and remove the form using the form helpers that generate add and remove links.

The following screenshot shows what the menu page looks like with a nested form to add items:



Creating user roles

Our aim here is to create a role-based authentication structure, where we will define various roles for the users. We will use a combination of the `rolify` gem to define roles and `cancan` gem, which includes methods to restrict users according to their roles.

Prepare for lift off

As in our previous projects, we have used the `devise` gem for creating the authentication system. We looked at authentication in *Project 1, A Social Recipe-sharing Website*, and have included it in every project ever since. So now, it is assumed that you will install and configure `devise` before you begin this step.

Engage thrusters

We will add the basics of the permissions framework in these steps:

1. Add the cancan gem and rolify gem to our Gemfile in order to use in conjunction with devise:

```
Gemfile
gem 'cancan'
gem 'rolify', '3.4'
```

2. Run the bundler and then generate the ability model to define authorizations:

```
:~/gotable$ rails g cancan:ability
      create  app/models/ability.rb
```

3. This will create a new file inside the models folder called ability. We will now generate the role model and related migrations using the rolify generator:

```
:~/gotable$ rails generate rolify:role
      create  app/models/role.rb
      insert  app/models/user.rb
      create  config/initializers/rolify.rb
      create  db/migrate/20130929082020_rolify_create_roles.rb
```

4. This will also generate an initializer file and insert the rolify method in the user model:

```
app/models/user.rb
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

5. Also, the role model has a reference to the join_table between users and roles:

```
app/model/role.rb
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users, :join_table => :users_roles
  belongs_to :resource, :polymorphic => true
  scopifyy
end
```

6. Once the role methods are generated, we will define the abilities of each user role:

```
app/models/ability.rb
class Ability
  include CanCan::Ability
  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.has_role? :admin
      can :manage, :all
    else
      can :read, Organization
      can :manage, Organization if user.has_role?(:owner, Organization)
      can :write, Organization, :id => Organization.with_role(:manager, user).map(&:id)
    end
  end
```

7. Let's try and add a role to our user. We will fire up our Rails console and call our last user in a variable:

```
1.9.3-p327 :001 > user = User.last
User Load (0.7ms)  SELECT `users`.* FROM `users` ORDER BY `users`.`id` DESC LIMIT 1
=> #<User id: 4, email: "admin@laboncafe.com", encrypted_password: "$2a$10$n7lb8ivkcnAFZZ5rSV0eOuhFWv7sU.HkrU0/OespLozh...", reset_password_token: nil, reset_password_sent_at: nil, remember_created_at: nil, sign_in_count: 8, current_sign_in_at: "2013-09-29 07:52:10", last_sign_in_at: "2013-09-29 07:51:47", current_sign_in_ip: "127.0.0.1", last_sign_in_ip: "127.0.0.1", created_at: "2013-09-29 04:22:11", updated_at: "2013-09-29 07:52:10", name: "laboncafe">
```

8. Once the variable value is set, we will add a role to it. As the user is the owner of the organization, we will add a role called owner:

```
1.9.3-p327 :003 > user.add_role "owner"
Role Load (0.7ms)  SELECT `roles`.* FROM `roles` WHERE `roles`.`name` = 'owner' AND `roles`.`resource_type` IS NULL AND `roles`.`resource_id` IS NULL ORDER BY `roles`.`id` ASC LIMIT 1
(0.6ms)  BEGIN
SQL (0.7ms)  INSERT INTO `roles` (`created_at`, `name`, `updated_at`) VALUES ('2013-09-29 08:38:26', 'owner', '2013-09-29 08:38:26')
(41.1ms)  COMMIT
Role Exists (0.5ms)  SELECT 1 AS one FROM `roles` INNER JOIN `users_roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE `users_roles`.`user_id` = 4 AND `roles`.`id` = 1 LIMIT 1
```

```
(0.4ms)  SELECT `roles`.id FROM `roles` INNER JOIN `users_
roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE `users_
roles`.`user_id` = 4
Role Load (0.4ms)  SELECT `roles`.* FROM `roles` WHERE
`roles`.`id` = 1 LIMIT 1
Role Load (0.5ms)  SELECT `roles`.* FROM `roles` INNER JOIN
`users_roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE
`users_roles`.`user_id` = 4
(0.2ms)  BEGIN
(0.3ms)  INSERT INTO `users_roles` (`user_id`, `role_id`)
VALUES (4, 1)
(45.9ms)  COMMIT
=> #<Role id: 1, name: "owner", resource_id: nil, resource_type:
nil, created_at: "2013-09-29 08:38:26", updated_at: "2013-09-29
08:38:26">
```

9. To see what the user can or cannot do, we will use the Ability model method:

```
1.9.3-p327 :002 > ability = Ability.new(user)
(0.7ms)  SELECT COUNT(*) FROM `roles` INNER JOIN `users_
roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE `users_
roles`.`user_id` = 4 AND (((roles.name = 'admin') AND (roles.
resource_type IS NULL) AND (roles.resource_id IS NULL)))
(1.2ms)  SELECT COUNT(*) FROM `roles` INNER JOIN `users_
roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE `users_
roles`.`user_id` = 4 AND (((roles.name = 'owner') AND (roles.
resource_type IS NULL) AND (roles.resource_id IS NULL)) OR
((roles.name = 'owner') AND (roles.resource_type = 'Organization')
AND (roles.resource_id IS NULL)))
(0.8ms)  SELECT COUNT(*) FROM `roles` INNER JOIN `users_
roles` ON `roles`.`id` = `users_roles`.`role_id` WHERE `users_
roles`.`user_id` = 4 AND `roles`.`name` = 'manager'
Organization Load (0.6ms)  SELECT `organizations`.* FROM
`organizations` INNER JOIN `roles` ON `roles`.resource_type =
'Organization' AND
(`roles`.resource_id IS NULL OR `roles`.resource_id =
`organizations`.id) WHERE (`roles`.name IN ('manager') AND
`roles`.resource_type = 'Organization') AND (`roles`.id IN (NULL)
AND ((resource_id = `organizations`.id) OR (resource_id IS NULL)))
1.9.3-p327 :003 > ability.can? :manage, :all
=> false
1.9.3-p327 :004 > ability.can? :manage, Organization
=> true
```

10. Load up the roles in the organization model so that these are applied to the organization model once it is initiated:

```
app/models/organization.rb
class Organization < ActiveRecord::Base
  resourcify
end
```

Objective complete – mini debriefing

In this task, we used the `rolify` gem to define different roles in the system. It provides a **DSL (domain-specific language)** that integrates with `cancan` and `devise` with ease. We also used the `cancan` gem to define the authorization and access for each role.

In order to do this, we first bundled our application with `cancan` and `rolify` gem. We then generated a model called `ability`. In the `ability` model, the `initialize` model directly hooks up to the `user` model:

```
def initialize(user)
  user ||= User.new # guest user (not logged in)
end
```

After this, we generated the `role` model using the `rolify` generator. The `rolify` generator depends on two methods: `scopify` and `resourcify`. The `scopify` method is defined in the `role` model. It loads the scopes (https://github.com/EppO/rolify/blob/master/lib/rolify/adapters/active_record/scopes.rb) and associates them with the `role` model. The `resourcify` method, once defined in a model, applies the roles to that model. Another advantage of using `rolify` is that it integrates well with `cancan`. In the `ability` model, we defined an authorization such that if the user has a role owner, he or she can manage an organization. The `manage` method allows a user to edit and delete a particular resource. Likewise, `read`, `write`, and other specific actions can be defined for users. If a user has this role, they will be allowed to perform only that action.

The `user.has_role?` method is a method from `rolify` that calls the `role` inside `cancan`.

Creating plans

The most important part of SaaS is a **multitier plan**. A lot of companies now keep one plan for the sake of simplicity. However, plans are created so that our application fits the requirement of companies of different sizes. This section will cover creating a plan and associating it with organizations.

Engage thrusters

We will create plans stepwise and associate it with our application resources:

1. Generate a plan model by running the Rails generator:

```
$ rails g model plan name:integer restaurants:integer price:float
tables:integer menu_items:integer storage:integer
```

2. Now run Rake db:migrate to add a plans table. Load seed data to the plans table:

```
db/seeds.rb
plans = [
    [ "Small", 1, 10, 20, 5, 10 ],
    [ "Medium", 5, 50, 50, 10, 30 ],
    [ "Large", 10, 100, 50, 50, 50 ]
]
plans.each do |name, restaurants, tables, menu_items, storage,
price|
    Plan.find_or_create( name: name, restaurants: restaurants,
tables:tables, menu_items:menu_items, storage:storage, price:price
)
end
```

3. Plans will now be added in our database.

4. We will now display these plans in the home page so that the user can compare them before signing up. For this, we will first create a home controller and home page:

```
Gotable$ rails g controller home index

app/controllers/home_controller.rb
class HomeController < ApplicationController
    def index
        @plans = Plan.all.to_a
    end
end

app/views/home/index.html.erb
<div class="row">
    <% @plans.each do |plan| %>
        <div class="col-sm-4">
            <div class="list-group">
                <a href="#" class="list-group-item active">
                    <%= plan.name %> - <%= number_to_currency(plan.
price) %> / Month
                </a>
            </div>
        </div>
    <% end %>
</div>
```

```
<a href="#" class="list-group-item"><%= plan.
restaurants %> Restaurant</a>
    <a href="#" class="list-group-item"><%= plan.tables %>
tables / Restaurant</a>
        <a href="#" class="list-group-item"><%= plan.menu_
items %> Menu Items / Restaurant</a>
            <a href="#" class="list-group-item"><%= plan.storage
%> GB storage</a>
                <%= link_to new_user_registration_path(:plan_id =>
plan.id), :class=>"list-group-item" do%><button class="btn btn-
success">Sign Up</button><%end%>
            </div>
        </div><!-- /.col-sm-4 -->
    <% end %>
</div>
```

5. Here, we are passing `plan_id` as a parameter in the link for **Sign Up**. After adding the **Plans** section, our home page will look like the following screenshot:

Plans		
Small - \$10.00 / Month	Medium - \$30.00 / Month	Large - \$50.00 / Month
1 Restaurant	5 Restaurant	10 Restaurant
10 tables / Restaurant	50 tables / Restaurant	100 tables / Restaurant
20 Menu Items / Restaurant	50 Menu Items / Restaurant	50 Menu Items / Restaurant
5 GB storage	10 GB storage	50 GB storage
Sign Up	Sign Up	Sign Up

6. We need to display the selected plan on our registration page. In order to do so, we will use pluck. As we need only the plan name, we will just use pluck to call the name of the plan with ID as the parameter. We will add the following in our `registrations_controller.rb` file:

```
app/helpers/application_helper.rb
module ApplicationHelper
  def plan_name(plan_id)
    plan_name = Plan.where(:id=> plan_id).pluck(:name).first
  end
end

app/controllers/devise/registrations_controller.rb
private
def update_sanitized_params
```

```
devise_parameter_sanitizer.for(:sign_up) {|u| u.permit(:name,  
:organization_name, :email, :password, :password_confirmation,  
:plan_id)}  
end
```

7. We will now display the plan in the form as follows:

```
app/views/devise/registrations/new.html.erb  
<h3>Plan Selected: <%= plan_name(params[:plan_id]) %></h3>
```

The screenshot shows a mobile application interface with a dark header bar containing the text "Go-table". The main content area has a light background. At the top, the title "Setup a user - Step 1" is displayed. Below the title are four input fields: "Domain name" (placeholder "Enter your domain name"), "Email" (placeholder "Email"), "Password" (placeholder "Password"), and "Password confirmation" (placeholder "Password Confirmation"). Underneath these fields, the text "Plan Selected: Medium" is shown. At the bottom of the screen, there are three buttons: "Sign up" (in a white box), "Sign in" (in a blue box), and "Forgot your password?" (in blue text).

8. Associate `plan` and `user`. In order to associate an organization to a plan, we will associate it via `user`. This will give a user the freedom to create multiple organizations and get billed for it through their own account. We will define them in our models:

```
models/plan.rb  
belongs_to :user
```

```
models/user.rb  
has_one :plan
```

9. We will now save plan IDs along with the user details. This is essential in order to keep a track of the subscription of each plan in accordance with the resources being used by each subscriber. In order to do so, we will first add `plan_id` to the user:

```
:~/gotable$ rails g migration add_plan_id_to_users plan_id:integer
      invoke  active_record
      create    db/migrate/20130930030144_add_plan_id_to_users.rb
```

10. The following is how the migration looks:

```
db/migrate/20130930030144_add_plan_id_to_users.rb
class AddPlanIdToUsers < ActiveRecord::Migration
  def change
    add_column :users, :plan_id, :integer
  end
end
```

11. Pass `plan_id` as a hidden field in the user signup form. We passed `plan_id` as parameter along with our link to plan in the home page:

```
app/views/devise/registrations/new.html.erb
<%= f.hidden_field :plan_id, :value => params[:plan_id] %>
```

Objective complete – mini debriefing

In the previous task, we created a structure for plans and defined them in the database. In order to do so, we created a model and table for plans and loaded some seed data into it. Seed data is used to add some default data to the application.

We also saw how to associate our `user` to a `plan`. In this way, we will be able to set up a monthly billing account for a particular plan for a specific user. We also saw the use of `pluck`:

```
plan_name = Plan.where(:id=> plan_id).pluck(:name).first
```

`Pluck` is an `ActiveRecord` query method that selects only a particular column; in this case it calls the column called `name`. We added some extra parameters to `devise`. In order to do so, we added a method called `update_sanitized_params` in our `registrations controller`. This method overrides the default `params` in `devise`:

```
def update_sanitized_params
  devise_parameter_sanitizer.for(:sign_up) {|u| u.permit(:name,
:organization_name, :email, :password, :password_confirmation, :plan_
id)}
end
```

We created a helper method to call the plan name in our application helper, which accepts `plan_id` as an argument. This is to display the plan name in views.

We also displayed the selected plan on the **Sign Up** page so that the user is clear about what he or she is signing up for.

Creating subdomains

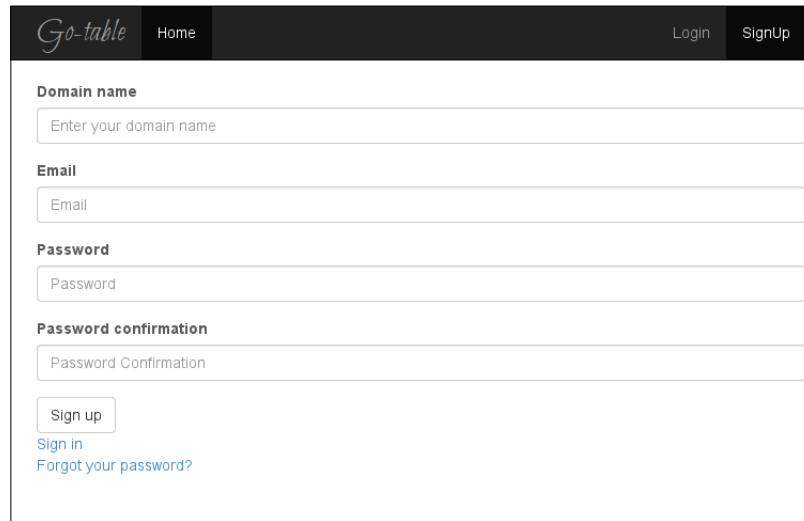
One of the main functions of a SaaS-based application is to provide the users with a separate area completely owned by them. This area is completely abstracted from others and is visible only to the owners and users of the organization. In this task, we will create subdomains and associate them with an organization. We will also explore the definition of a **concern** in detail and how can it be used to create reusable code components. We will use Tim Pope's solution (<http://tbaggery.com/2010/03/04/smack-a-ho-st.html>) of extending a domain name called local virtual host (lvh.me) in order to make subdomains work on our localhost.

Engage thrusters

Let us create subdomains for our application users:

1. We will first save the domain as a part of our **SignUp** form:

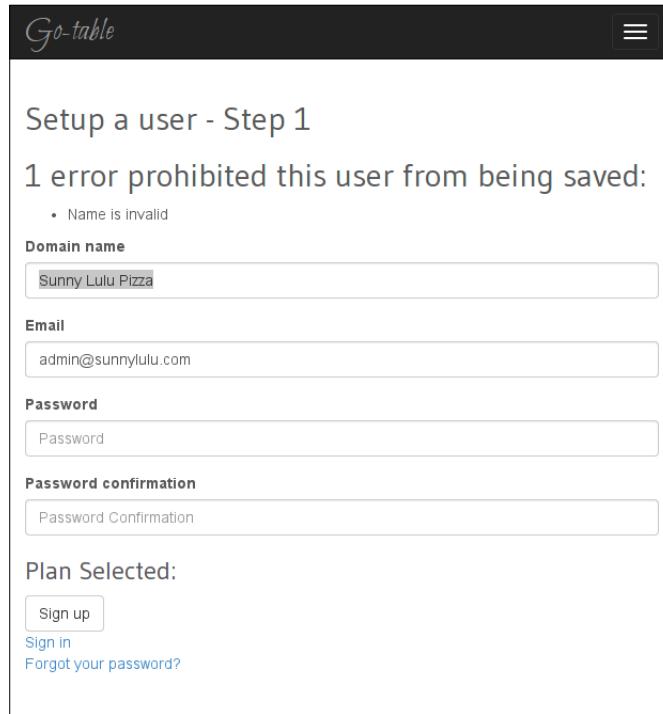
```
app/views/devise/registrations/new.html.erb
<div class="form-group">
  <%= f.label 'domain name' %><br />
  <%= f.text_field :domain_name, :autofocus => true,
  :class=>"form-control", :placeholder => "Domain Name" %>
</div>
```



The screenshot shows a web browser window with a dark header bar. The header contains the logo 'Go-table' on the left, and 'Home', 'Login', and 'SignUp' buttons on the right. The main content area is a sign-up form. It has four input fields labeled 'Domain name', 'Email', 'Password', and 'Password confirmation'. Below these fields are three small links: 'Sign up' (in a button-like style), 'Sign in', and 'Forgot your password?'. The entire form is contained within a light-colored box.

2. Domain names should not have spaces between the words. In order to avoid these, we will add a validation to the user model:

```
app/models/user.rb
validates :name, presence: true, format: { without:
/^((http|https)\/\/)[a-z0-9]*(\.?[a-z0-9]+)\.[a-z]{2,5}(:[0-9]
{1,5})?(\.?)?$/ix, multiline: true }
```



3. In order to create subdomains, we need a class that passes the value of the request and matches the format of a subdomain. As www is not considered a valid subdomain rule, we will check for this and make it nil:

```
lib/subdomain.rb
class Subdomain
  def self.matches?(request)
    case request.subdomain
    when 'www', ''
      false
    else
      true
    end
  end
end
```

4. Once this method is set up, we will make a call inside our controller. In order to make only the authenticated user log in, we will create dashboard_controller and add an authenticate_user! filter to it. Dashboard is the page where we can see our activity stream:

```
gotable$ rails g controller dashboard  
controllers/dashboard_controller.rb  
  before_filter :authenticate_user!
```

5. We will also check if the user belongs to that subdomain or not:

```
app/controllers/dashboard_controller.rb  
  before_filter :load_subdomain  
  def show  
    @user = User.where(:name => request.subdomain).first || not_found  
    @user.organizations.each do |o|  
      @organization_name = o.name  
    end  
  end  
  def not_found  
    raise ActionController::RoutingError.new('User Not Found')  
  end  
  def load_subdomain  
    @user = User.where(:domain_name => request.subdomain).first  
  end
```

6. Wire this concern to the route. We will pass our subdomain class as a constraint in order to check the subdomain format as soon as the request is made. Also, we will see if the user has been authenticated or not and based on this, we will redirect him or her to the respective organization's dashboard:

```
config/routes.rb  
  authenticated do  
    get '/' => 'dashboard#show', :constraints => Subdomain, :as => 'dashboard'  
  end
```

7. Create a method in order to first check if there is a value of subdomain supplied or not. If it is present, it will append the subdomain, domain, and port. We also check for the presence of a hash key called subdomain. If the key is present, it will add the value of the host to the value of the with_subdomain method:

```
app/controllers/concern/subdomain.rb  
module Concerns  
  module Url
```

```
extend ActiveSupport::Concern
def with_subdomain(subdomain)
  subdomain = (subdomain || "")
  subdomain += "." unless subdomain.empty?
  [subdomain, request.domain, request.port_string].join
end
def url_for(options = nil)
  if options.kind_of?(Hash) && options.has_key?(:subdomain)
    options[:host] = with_subdomain(options.delete(:subdomain))
  end
  super
end
end
```

8. In order to execute this Url manipulation, we will need to include this in the application controller and extend it:

```
controllers/application_controller.rb
class ApplicationController < ActionController::Base
  include Concerns::Url
end
```

9. Now that domains are there, we will have to ensure that the sessions of each subdomain are different from the other. By adding a :domain => :all method, we will have a different session store for each subdomain:

```
config/initializers/session_store.rb
Gotable::Application.config.session_store :cookie_store, key:'_gotable_session', :domain => :all
```

Objective complete – mini debriefing

At the end of this section, we have successfully created subdomains, abstracted their sessions, and made sure all the redirects are in place. We first validated our domain_name with a regex. In Rails 4, the multiline option is mandatory for the regex to work as it contains anchors such as the dollar sign:

```
format: { without: /^((http|https):\/\/)[a-z0-9]*(\.? [a-z0-9]+)\. [a-z]{2,5}(:[0-9]{1,5})?(\?/. )?$/ix, multiline: true }
```

The preceding regex matches the format of the domain with numbers, letters, and also the protocol used in the URL. We created a `Subdomain` class, which we used in our route as a constraint. This constraint will make the `www` request on a subdomain `nil`, as `www` is an invalid request for a subdomain:

```
class Subdomain
  def self.matches?(request)
    case request.subdomain
    when 'www', '', nil
      false
    else
      true
    end
  end
end
:constraints => Subdomain
```

We then added a rule for searching the domain once the request is made. This will ensure that the domain is present in the database:

```
before_filter :load_subdomain
def load_subdomain
  @user = User.where(:name => request.subdomain).first
end
```

In case it was not found, we sent a custom routing error:

```
def not_found
  raise ActionController::RoutingError.new('User Not Found')
end
```

We also restricted all our work according to the domain owned by a user. The session is owned by a user and an organization. In order to do so, we used the concerns pattern. In Rails 4, `concerns` comes as a default folder. It is used to define the code that is reusable in different contexts across the application. `Subdomain` is defined as a module with a collection of classes and methods. This module can then be included in any controller or model of our choice. We defined a controller concern to request and extract the subdomain from the URL. We then ensured that this concern is loaded across the entire application. It is important to understand that sometimes the same concern can be used in different ways.

If we have categories across multiple models, we can create a concern for it and load it across different models. Lastly, we separated out sessions for each subdomain because each organization has a different one:

```
Gotable::Application.config.session_store :cookie_store, key: '_  
gotable_session', :domain => :all
```



Adding multitenancy and reusable methods

We have already set up subdomains in order to create separate areas for each organization. However, we have to make sure that the users from one organization do not see data from another organization. A clear separation of this data visibility is called multitenancy. The concept can be compared to renting out apartments to multiple tenants. We will add multitenancy in our application by adding a simple method in our model concern.

Engage thrusters

1. Next up, we will create the multitenant model of our application. Let's see how. We will first create a separate class to handle the tenants. This class will handle all the code related to tenancy:

```
:~/gotable/app/models$ touch tenant.rb
```

2. Tenant is a simple ruby class. We will first initialize a user object and extend it in the subsequent steps. We will also pass roles to the user object as we will also check the visibility according to roles:

```
app/models/tenant.rb  
class Tenant  
  def initialize user  
    @user = user  
  end  
  private  
  def admin?  
    @user.has_role? "admin"  
  end  
  def owner?
```

```
    @user.has_role? "owner"
  end
end
```

3. Now, we will show only the restaurants that are available to a particular organization for a particular role. For this, we will first check for the role associated with the user object. Then, we will create a scope to find the restaurants with a particular organization_id associated with them:

```
App/models/tenant.rb
def restaurants
  admin? ? Restaurant.all.all : Restaurant.where('organization_
id = ?', user.organizations.first.id).all
end
```

4. We will then set this as a filter for our entire application. We will pass current_user from devise as the user object:

```
controllers/application_controller.rb
before_filter :enable_tenant
def enable_tenant
  @current_tenant ||= Tenant.new(current_user.organization)
end
```

5. Finally, make a call in the controller and call the Tenant class before finding the value of restaurants. In this way, if a user is the owner of a restaurant, only the restaurants owned by him or her are visible to them:

```
app/views/controllers/restaurants_controller.rb
def index
  if params[:id].present?
    @restaurants = @current_tenant.restaurants.find(params[:id])
  end
end
```

Objective complete – mini debriefing

Quoting from Wikipedia (<https://en.wikipedia.org/wiki/Multitenancy>):

Multitenancy refers to a principle in software architecture where a single instance of the software runs on a server, serving multiple client-organizations (tenants).

In order to achieve this, we need to create tenants. Tenants have been extracted into a separate class which checks for the user with the role owner or admin.

We then checked for restaurants associated to a particular tenant. We first queried for the owner role as the organization is associated to a user. If the user is an admin, we return all the restaurants. The colon (:) here represents the `if else` condition. If the user is not an admin, we chained the query for finding restaurants over the restaurant scope. The following query finds the current user according to the organization ID for all restaurants:

```
admin? ? Restaurant.all.all : Restaurant.where('organization_id = ?',
  user.organizations.first.id).all
```

`Rails.scoped` was a method used earlier to define a chained query format. However, with Rails 4, it has been deprecated and removed. Instead, `Restaurant.all` behaves in the same way. In order to achieve what we previously did in `Restaurant.all(select * from restaurants)`, we now need to either do `Restaurant.all.to_a` or `Restaurant.all.all`. We then instantiated a new `Tenant` instance as soon a user logs in to create `current_tenant`:

```
@current_tenant = Tenant.new(current_user)
```

We separated the data of each user depending on their `organization_id` and their role in that organization. This will help us set up a clear policy framework based on the roles of different users and their rights. This will also help us in tracking things such as billing and checking the limits of plans because all of this is calculated as per the organization.

Creating a monthly payment model, adding a free trial plan, and generate a monthly bill

The most important part of a SaaS application is the ability to bill every month. A lot of applications also give out free trials in order to allow the user to actually try out these applications before they are billed for it. This section will cover how to generate a monthly bill. However, we will not cover an actual payment gateway in this section.

Engage thrusters

1. We will add the monthly billing code now. Add the credit card details to `users` table, update the parameters, and validate them:

```
controllers/devise/registrations_controller.rb
def update_sanitized_params
  devise_parameter_sanitizer.for(:sign_up) {|u| u.permit(:name,
:organization_name, :email, :password, :password_confirmation,
:plan_id, :active, :first_name, :last_name)}
end
```

2. In order to check whether we have a free trial, we need to get the time difference between the date of joining of the user and the current date. A free trial is valid only if the difference is less than a month. This is because we want the free trial to last for one month:

```
app/models/user.rb
def date_difference(date1,date2)
  month = (date2.year - date1.year) * 12 + date2.month - date1.
month - (date2.day >= date1.day ? 0 : 1)
end
```

3. The preceding code will return a value in months and will take two dates as parameters. In order to check whether we have a free trial, we will use the date difference method we just created. We will return a value `true` or `false` based on our condition. If it is `false`, then we will generate an error message about the trial being ended:

```
app/models/user.rb
def free_trial
  month = date_difference(self.created_at, Date.today)
  if month >= 1
    return false
  errors.add(:trial_end, "Your Free Trial Has Ended, please
select from a plan")
  elsif month < 1
    return true
  end
end
```

4. At this point, we will generate a model to record transactions and save them in our database for future references:

```
$ rails g migration transactions user_id:integer status:boolean
created_at:datetime updated_at:datetime amount:decimal first_
name:string last_name:string
```

The following code shows what our `transactions` table looks like:

```
create_table "transactions", force: true do |t|
  t.integer  "user_id"
  t.boolean   "status"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.decimal   "amount",      precision: 10, scale: 0
  t.string    "first_name"
  t.string    "last_name"
end
```

5. We will also check if there is an outstanding amount or not. We will check the last transaction date and find the date difference. If it is over a month, we will generate an error message:

```
app/models/user.rb
def if_amount_pending
    trial = self.free_trial
    if (trial == false && self.active?)
        last_transaction = Transaction.where(:user_id => self.id).last
        if date_difference(last_transaction.created_at, Date.today) >= 1
            errors.add(:pending, "You have an outstanding invoice, kindly pay at the earliest.")
            true
        end
    end
end
```

6. Before charging the card, we will double check if it is valid or not. Sometimes it so happens that the card is valid at the signup and expires later. In such cases, we need to check for the card validity every time it is charged:

```
app/models/user.rb
def credit_card_valid
    date = Date.today
    year = self.year
    month = self.month
    expiry_month = (date.year - year) * 12 + date.month - month
    if expiry_month > 6
        true
    elsif expiry_month < 6
        false
        errors.add(:transaction, "Credit Card not valid")
    end
end
```

7. In this case, we are checking if the expiry date of a card is less than six months or not.
8. Finally, we will put all this together to generate a `transaction` object. The condition needed to change a user will be he or she has a valid credit card, has finished the trial period, has an active account, and has an outstanding amount to be paid:

```
app/models/user.rb
def charge_credit_card
    trial = self.free_trial
```

```
amount_pending = self.amount_pending
credit_card_valid = self.credit_card_valid
if (trial == false && self.active? && amount_pending == true
&& credit_card_valid == true)
  plan_id = self.plan_id
  plan = Plan.where(:id => plan_id).first
  transaction = Transaction.new
  transaction.attributes(user_id: self.id, first_name: self.
first_name, last_name: self.last_name, card_type: self.card_type,
card_number: self.card_number, cvv: self.cvv, month: self.month,
year: self.year, amount: plan.price)
  if transaction.save!
    transaction.status = true
    transaction.update
  else
    transaction.status = false
    transaction.update
    errors.add(:transaction, "Credit Card Could not be
Charged")
  end
end
end
```

Objective complete – mini debriefing

This section is highly subjective and can also be deemed as optional. It applies to a lot of use cases where billing and invoicing is separate from the charges of the credit card. A lot of this code, especially, the way transactions and credit card details are handled, may significantly differ from what we have covered here. A lot of payment gateways store the credit card details so that we don't have to handle it for PCI compliance. There are other gateways as well that tokenize the credit card details.

Here we saw how to create a monthly charge method and check whether there are any free trials in our plans. Although this code is very generic, it can be used readily with custom methods that are specific to payment gateways written right inside these methods. Also, errors, PCI compliance techniques, the structure of the object, and validations are mostly specific to the payment gateway and hence not covered. We also looked at transactions, which are important because we need to create an object for sending to the payment gateway regardless of the provider. So, the `transaction` model we looked at in the preceding task serves a dual purpose of sending the information to the payment gateway and also recording the transaction in our database.

Exporting data to a CSV format

We often need to transfer data between different systems. Also, sometimes we need to send data to different people in different formats, whom we may or may not want to give our system's access directly. In order to do so, we generally export the data into formats that are commonly readable. One of the most common formats is CSV, or the comma separated values format. It's quick to export because it's a text-only format and most files are small in size. Also, it is compatible with most text editors.

Engage thrusters

The final task contains steps to add the **Export to CSV** functionality. Let's go ahead and add them:

1. Ruby natively supports the CSV mime type, so it is quite easy and quick to get started with. As it is available as a module of Ruby, we will make a call on it in our application.rb in order to load it for our application:

```
config/application.rb
require File.expand_path('../boot', __FILE__)
require 'rails/all'
```

2. CSV is like just another format for rendering and is similar to XML and HTML:

```
app/controllers/restaurants_controller.rb
require 'csv'
def export_menus
  @menus = Menu.where(:restaurant_id => params[:restaurant_id])
  respond_to do |format|
    format.html
    format.csv { render text: @menus.export_to_csv }
  end
end
```

3. However, there is no `export_to_CSV` method as yet. We will generate an array, loop over all the column names, and convert them to a CSV file:

```
app/models/menu.rb
require 'csv'
def self.export_to_CSV
  CSV.generate do |CSV|
    CSV << column_names
    all.each do |menu|
```

```
    CSV << menu.attributes.values_at(*column_names)
  end
end
end
```

4. In order to export, we need a link to do so. This link will generate and save a CSV file:

```
config/routes.rb
resources :restaurants do
  collection do
    get 'export_menus'
  end
end
app/views/restaurants/show.html.erb
<%= link_to "Export to CSV", export_menus_restaurants_
path(format: "csv", :restaurant_id => @restaurant.id),
:class=>"btn btn-default" %>
```

Objective complete – mini debriefing

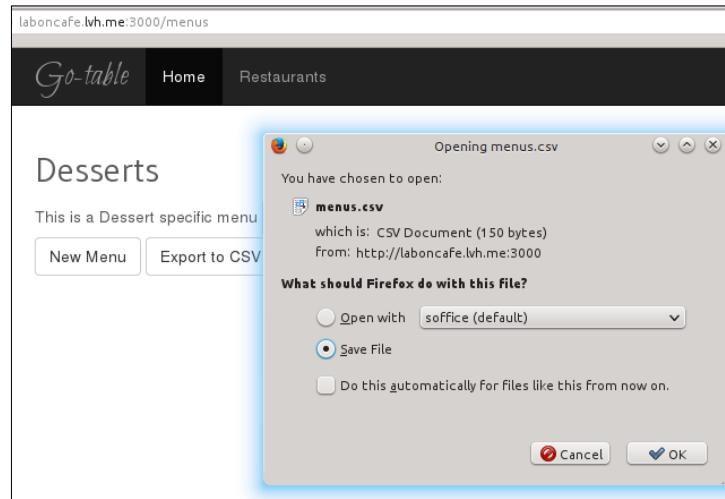
In this section, we saw how to generate and export a CSV file from records in our database. This helps a lot in several enterprise applications. We may extend this to other data formats as well; for example, Excel. This helps in interoperability of various systems. For example, there is an existing application Xero (xero.com) for accounting, which works in tandem with our application; this could be a very useful feature.

The CSV mime type is a part of the `ActionController::Renders` module. In order to use this in a controller, we included it at the top of the controller.

We then queried and copied all the values of the menus into a single array. This array of menu objects is then rendered as CSV.

Just like we define the response format in our application as HTML, JSON, or XML, CSV can also be defined as a rendering format.

As soon as we click on **Export to CSV**, we will get a prompt to save it as shown in the following screenshot. We extended our resource by adding a collection route to it. This collection route calls the `export_menus` format in order to call the `export` method for CSV. We can export the menu data as shown in the following screenshot:



Mission accomplished

In this project, we created a simple SaaS-based application and saw how it is structured. We focused more on concepts of SaaS-based applications such as multitenancy, abstraction, roles and policy framework, and customizing the signup process into a wizard. We solved various issues related to the separation of data and privacy for each user. We also created plans and ways to set up a basic system with plans and pricing. Finally, we learned how to export data from the system for interoperability and work with existing applications.

Hotshot challenges

Of course, we need to enhance what we just learned. The following are some exercises that will help us do so:

- ▶ Use Stripe to add a payment gateway
- ▶ Add a method to cancel the billing of a user and discontinue it
- ▶ Create the export to spreadsheet option along with CSV
- ▶ Add errors to the payment gateway and display them in the frontend
- ▶ Add integration tests for testing subdomains

Project 5

Building a Customizable Content Management System

Content is the backbone of the Internet. A **Content Management System (CMS)** is essentially a software that helps you to easily and effectively manage the content of a website or a web application. There are several perspectives on CMS, with Drupal, Joomla!, and WordPress being the really popular ones. However, people still build tailor-made CMSes, because they want something that fits their needs exactly.

Mission briefing

This project deals with the creation of a Content Management System. This system will consist of two parts:

- ▶ A backend that helps to manage content, page parts, and page structure
- ▶ A frontend that displays the settings and content we just entered

We will start this by creating an admin area and then create page parts with types. Page parts, which are like widgets, are fragments of content that can be moved around the page. Page parts also have types; for example, we can display videos in our left column or display news. So, the same content can be represented in multiple ways. For example, news can be a separate page as well as a page part if it needs to be displayed on the front page. These parts need to be enabled for the frontend. If enabled, then the frontend makes a call on the page part ID and renders it in the part where it is supposed to be displayed. We will do a frontend markup in Haml and Sass.

The following screenshot shows what we aim to do in this project:

The screenshot displays the 'Admin Dashboard' of the Helios CMS. At the top, there is a navigation bar with tabs for 'Helios CMS', 'Admin', 'Page', 'Page Parts', and 'Site Details'. On the right side of the navigation bar, it shows the email address 'saurabh.a.bhatia@gmail.com' and a 'Logout' button. Below the navigation bar, the main area is titled 'Admin Dashboard'. It contains two main sections: 'Pages' and 'Page Parts'.
Pages Section: This section has a header with three columns: 'Title', 'Parent', and 'Type'. It lists two items:

Title	Parent	Type
Home	Home	
About	Home	

Page Parts Section: This section has a header with three columns: 'Title', 'Type', and 'Created By'. It lists two items:

Title	Type	Created By
YouTube Channel		
Bulletin		

Why is it awesome?

Everyone loves to get a CMS built from scratch that is meant to suit their needs really closely. We will try to build a system that is extremely simple as well as covers several different types of content. This system is also meant to be extensible, and we will lay the foundation stone for a highly configurable CMS. We will also spice up our proceedings in this project by using MongoDB instead of a relational database such as MySQL.

At the end of this project, we will be able to build a skeleton for a very dynamic CMS.

Your Hotshot objectives

While building this application, we will have to go through the following tasks:

- ▶ Creating a separate admin area
- ▶ Creating a CMS with the ability of handling different types of content pages
- ▶ Managing page parts
- ▶ Creating a Haml- and Sass-based template
- ▶ Generating the content and pages
- ▶ Implementing asset caching

Mission checklist

We need to install the following software on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MongoDB
- ▶ Bootstrap 3.0
- ▶ Haml
- ▶ Sass
- ▶ Devise
- ▶ Git
- ▶ A tool for mockups
- ▶ jQuery
- ▶ ImageMagick and RMagick
- ▶ Memcached

Creating a separate admin area

Since we have used devise for all our projects so far, we will use the same strategy in this project. The only difference is that we will use it to log in to the admin account and manage the site's data. This needs to be done when we navigate to the URL/admin. We will do this by creating a namespace and routing our controller through the namespace. We will use our default application layout and assets for the admin area, whereas we will create a different set of layout and assets altogether for our frontend. Also, before starting with this first step, create an admin role using CanCan and rolify and associate it with the user model. We are going to use memcached for caching, hence we need to add it to our development stack. We will do this by installing it through our favorite package manager, for example, apt on Ubuntu:

```
sudo apt-get install memcached
```



Memcached is a key-value cache store that stores small fragments of data.

Prepare for lift off

In order to start working on this project, we will have to first add the `mongoid` gem to `Gemfile`:

```
Gemfile
gem 'mongoid'4', github: 'mongoid/mongoid'
```

Bundle the application and run the `mongoid` generator:

```
rails g mongoid:config
```

You can edit `config/mongoid.yml` to suit your local system's settings as shown in the following code:

```
config/mongoid.yml
development:
  database: helioscms_development
  hosts:
    - localhost:27017
  options:
  test:
  sessions:
    default:
      database: helioscms_test
      hosts:
        - localhost:27017
      options:
        read: primary
        max_retries: 1
        retry_interval: 0
```

We did this because `ActiveRecord` is the default **Object Relationship Mapper (ORM)**. We will override it with the `mongoid` **Object Document Mapper (ODM)** in our application. Mongoid's configuration file is slightly different from the `database.yml` file for `ActiveRecord`. The session's rule in `mongoid.yml` opens a session from the Rails application to MongoDB. It will keep the session open as long as the server is up. It will also open the connection automatically if the server is down and it restarts after some time. Also, as a part of the installation, we need to add `Haml` to `Gemfile` and bundle it:

```
Gemfile
gem 'haml'
gem "haml-rails"
```

Engage thrusters

Let's get cracking to create our admin area now:

1. We will first generate our dashboard controller:

```
rails g controller dashboard index
create  app/controllers/dashboard_controller.rb
route  get "dashboard/index"
invoke  erb
create    app/views/dashboard
create    app/views/dashboard/index.html.erb
invoke  test_unit
      create    test/controllers/dashboard_controller_test.rb
invoke  helper
create    app/helpers/dashboard_helper.rb
invoke  test_unit
create    test/helpers/dashboard_helper_test.rb
invoke  assets
invoke  coffee
      create      app/assets/javascripts/dashboard.js.coffee
invoke  scss
      create      app/assets/stylesheets/dashboard.css.scss
```

2. We will then create a namespace called `admin` in our `routes.rb` file:

```
config/routes.rb
namespace :admin do
  get '', to: 'dashboard#index', as: '/'
end
```

3. We have also modified our dashboard route such that it is set as the root page in the `admin` namespace.

4. Our dashboard controller will not work anymore now. In order for it to work, we will have to create a folder called `admin` inside our controllers and modify our `DashboardController` to `Admin::DashboardController`. This is to match the `admin` namespace we created in the `routes.rb` file:

```
app/controllers/admin/dashboard_controller.rb
class Admin::DashboardController < ApplicationController
  before_filter :authenticate_user!

  def index
  end
end
```

5. In order to make the login specific to the admin dashboard, we will copy our devise/sessions_controller.rb file to the controllers/admin path and edit it. We will add the admin namespace and allow only the admin role to log in:

```
app/controllers/admin/sessions_controller.rb
class Admin::SessionsController < ::Devise::SessionsController

  def create
    user = User.find_by_email(params[:email])
    if user && user.authenticate(params[:password]) &&
       user.has_role? "admin"
      session[:user_id] = user.id
      redirect_to admin_url, notice: "Logged in!"
    else
      flash.now.alert = "Email or password is invalid /
        Only Admin is allowed"
    end
  end
end
```

Objective complete – mini debriefing

In the preceding task, after setting up devise and CanCan in our application, we went ahead and created a namespace for the admin.

In Rails, the namespace is a concept used to separate a set of controllers into a completely different functionality. In our case, we used this to separate out the login for the admin dashboard and a dashboard page as soon as the login happens. We did this by first creating the admin folder in our controllers. We then copied our Devise sessions controller into the admin folder. For Rails to identify the namespace, we need to add it before the controller name as follows:

```
class Admin::SessionsController < ::Devise::SessionsController
```

In our route, we defined a namespace to read the controllers under the admin folder:

```
namespace :admin do
  end
```

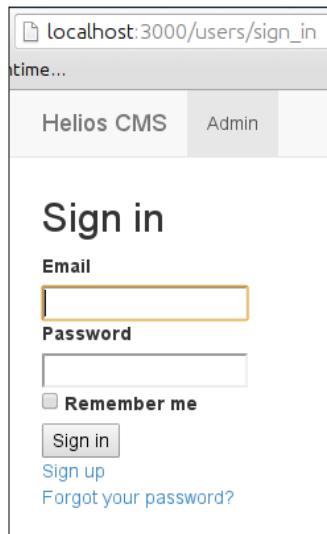
We then created a controller to handle dashboards and placed it within the admin namespace:

```
namespace :admin do
  get '', to: 'dashboard#index', as: '/'
end
```

We made the dashboard the root page after login. The route generated from the preceding definition is `localhost : 3000 /admin`. We have already seen how to add roles in our previous project, hence we assumed here that the admin role can be created. We ensured that if someone tries to log in by clicking on the admin dashboard URL, our application checks whether the user has a role of admin or not. In order to do so, we used `has_role` from rolify along with `user.authenticate` from devise:

```
if user && user.authenticate(params[:password]) && user.has_role?  
  "admin"
```

This will make devise function as part of the admin dashboard. If a user tries to log in, they will be presented with the devise login page as shown in the following screenshot:



The screenshot shows a web browser window with the URL `localhost:3000/users/sign_in` in the address bar. The page title is "Sign in". At the top, there is a navigation bar with "Helios CMS" and "Admin" tabs. Below the title, there are fields for "Email" and "Password", a "Remember me" checkbox, and a "Sign in" button. There are also links for "Sign up" and "Forgot your password?".

After logging in successfully, the user is redirected to the link for the admin dashboard:



Creating a CMS with the ability to create different types of pages

A website has a variety of types of pages, and each page serves a different purpose. Some are limited to contact details, while some contain detailed information about the team. Each of these pages has a title and body. Also, there will be subpages within each navigation; for example, the **About** page can have **Team**, **Company**, and **Careers** as subpages. Hence, we need to create a parent-child self-referential association. So, pages will be associated with themselves and be treated as parent and child.

Engage thrusters

In the following steps, we will create page management for our application. This will be the backbone of our application.

1. Create a model, view, and controller for page. We will have a very simple page structure for now. We will create a page with title, body, and page type:

```
app/models/page.rb
class Page
  include Mongoid::Document

  field :title, type: String
  field :body, type: String
  field :page_type, type: String

  validates :title, :presence => true
  validates :body, :presence => true

  PAGE_TYPE= %w(Home News Video Contact Team Careers)
end
```

2. We need a home page for our main site. So, in order to set a home page, we will have to assign it the type home. However, we need two things from the home page: it should be the root of our main site and the layout should be different from the admin. In order to do this, we will start by creating an action called `home_page` in `pages_controller`:

```
app/models/page.rb
scope :home, ->{where(page_type: "Home")}

app/controllers/pages_controller.rb
def home_page
```

```
@page = Page.home.first rescue nil  
  
    render :layout => 'page_layout'  
end
```

3. We will find a page with the `home` type and render a custom layout called `page_layout`, which is different from our application layout. We will do the same for the `show` action as well, as we are only going to use `show` to display the pages in the frontend:

```
app/controllers/pages_controller.rb  
  def show  
    render :layout => 'page_layout'  
  end
```

4. Now, in order to effectively manage the content, we need an editor. This will make things easier as the user will be able to style the content easily using it. We will use `ckeditor` in order to style the content in our application:

```
Gemfile  
gem "ckeditor", :github => "galetahub/ckeditor"  
gem 'carrierwave', :github => "jnicklas/carrierwave"  
  
gem 'carrierwave-mongoid', :require => 'carrierwave/mongoid'  
  
gem 'mongoid-grid_fs', github: 'ahoward/mongoid-grid_fs'
```

5. Add the `ckeditor` gem to `Gemfile` and run `bundle install`:

```
helioscms$ rails generate ckeditor:install --orm=mongoid  
--backend=carrierwave  
  
create config/initializers/ckeditor.rb  
route mount Ckeditor::Engine => '/ckeditor'  
create app/models/ckeditor/asset.rb  
create app/models/ckeditor/picture.rb  
create app/models/ckeditor/attachment_file.rb  
      create app/uploaders/ckeditor_attachment_file_uploader.  
rb
```

6. This will generate a carrierwave uploader for CKEditor, which is compatible with mongoid.
7. In order to finish the configuration, we need to add a line to `application.js` to load the `ckeditor` JavaScript:

```
app/assets/application.js  
//= require ckeditor/init
```

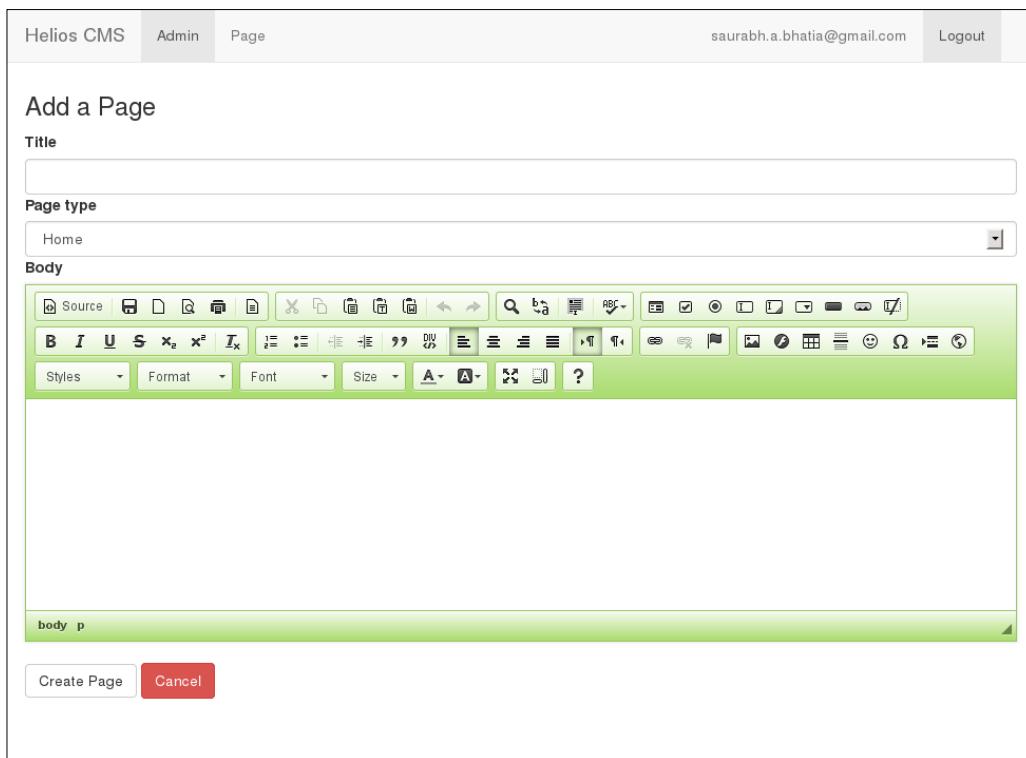
8. We will display the editor in the body as that's what we need to style:

```
views/pages/_form.html.haml
.field
  = f.label :body
  %br/
  = f.cktext_area :body, :rows => 20, :ckeditor => { :uiColor =>
    "#AADC6E", :toolbar => "mini" }
```

9. We also need to mount the ckeditor in our routes.rb file:

```
config/routes.rb
mount Ckeditor::Engine => '/ckeditor'
```

10. The editor toolbar and text area will be generated as seen in the following screenshot:



11. In order to display the content on the index page in a formatted manner, we will add the html_safe escape method to our body:

```
views/pages/index.html.haml
%td= page.body.html_safe
```

12. The following screenshot shows the index page after the preceding step:

The screenshot shows the 'Manage Pages' section of the Helios CMS. At the top, there are tabs for 'Helios CMS', 'Admin', and 'Page'. On the right, it shows the email 'saurabh.a.bhatia@gmail.com' and a 'Logout' button. Below the tabs, the title 'Manage Pages' is displayed. A table lists two pages: 'About' and 'Home'. Each row contains the page title, its content, and three action buttons (edit, preview, delete). An 'Add Page' button is located at the bottom left.

Title	Body	Action	Action	Action
About	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis nec odio eget lorem congue imperdiet. Nam vulputate massa luctus molestie sagittis. Duis sagittis diam elit, egestas pharetra sem venenatis vita. Vivamus vestibulum suscipit metus, nec porttitor lectus condimentum a. Praesent in ipsum dolor. Integer interdum rhoncus dolor eleifend fermentum. Sed sit amet tellus molestie, faucibus sapien et, ultrices nisi. Donec ultricies turpis venenatis ipsum faucibus tincidunt. In in felis dolor. Aenean sed tortor luctus sem varius facilisis. In pretium urna id pellentesque fermentum. Ut hendrerit imperdiet venenatis. Phasellus ultrices velit non metus sagittis faucibus.</p> <p>Curabitur consequat mauris quis vestibulum faucibus. Praesent quam eros, lacinia vitae molestie sed, lacinia vitae enim. Donec aliquet risus eget dignissim iaculis. Donec elementum quam leo, quis tempor nisi scelerisque quis. Morbi venenatis auctor dui, eget sollicitudin mauris blandit auctor. Nulla fringilla nibh id consectetur hendrerit. Vivamus congue, ipsum id pretium auctor, ipsum orci imperdiet libero, vitae consequat tellus turpis at erat. Donec vitae neque imperdiet, pellentesque mauris et, bibendum leo. In vulputate massa non erat fringilla accumsan. Nulla luctus ligula vel lorem sodales sollicitudin. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sed magna lacinus. Curabitur semper pharetra purus eu placerat. Fusce ut hendrerit est.</p>			
Home	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur vel dolor id nulla blandit euismod. Morbi a pulvinar diam, at congue purus. Maecenas eget enim nec arcu ornare malesuada eu ut libero. Mauris pharetra dui metus, ac fringilla tellus fermentum nec. Sed semper turpis ut fringilla vulputate. Ut placerat rutrum mattis. Vestibulum in ante ac mi ultricies volutpat ut non tortor. Nunc hendrerit justo lacinus, sed blandit odio eleifend vel. Vivamus nunc enim, hendrerit a iaculis sit amet, ultrices sit amet augue. Duis eget lorem id lorem convallis semper non id turpis. Vestibulum consequat elementum risus. Sed nec pretium sem. Ut odio eros, ultricies in pretium vitae, condimentum non purus. Suspendisse ultrices augue quam, ornare auctor felis blandit quis. Nullam nisi tortor, pretium in imperdiet ac, convallis non tortor.</p> <p>Etiam eleifend, lectus sed eleifend malesuada, lorem magna loboris justo, vel iaculis mauris justo ut lorem. Morbi laoreet lacinus ut est faucibus vestibulum. Vestibulum sodales leo vel enim tincidunt fringilla. Duis quis augue elementum, tempor quam id, scelerisque orci. Suspendisse cursus tortor et pulvinar commodo. Nam aliquet eleifend leo nec volutpat. Duis sagittis diam eu tellus hendrerit adipiscing. Aliquam imperdiet elit neque. Vivamus eleifend vel est ut semper.</p>			

Add Page

13. At this point, we can manage the content using pages. However, in order to add nesting, we will have to create a parent-child structure for our pages. In order to do so, we will have to first generate a model to define this relationship:

```
helioscms$ rails g model page_relationship
```

14. Inside the `page_relationship` model, we will define a two-way association with the `page` model:

```
app/models/page_relationship.rb
class PageRelationship
  include Mongoid::Document
  field :parent_id, type: Integer
  field :child_id, type: Integer

  belongs_to :parent, :class_name => "Page"
  belongs_to :child, :class_name => "Page"
end
```

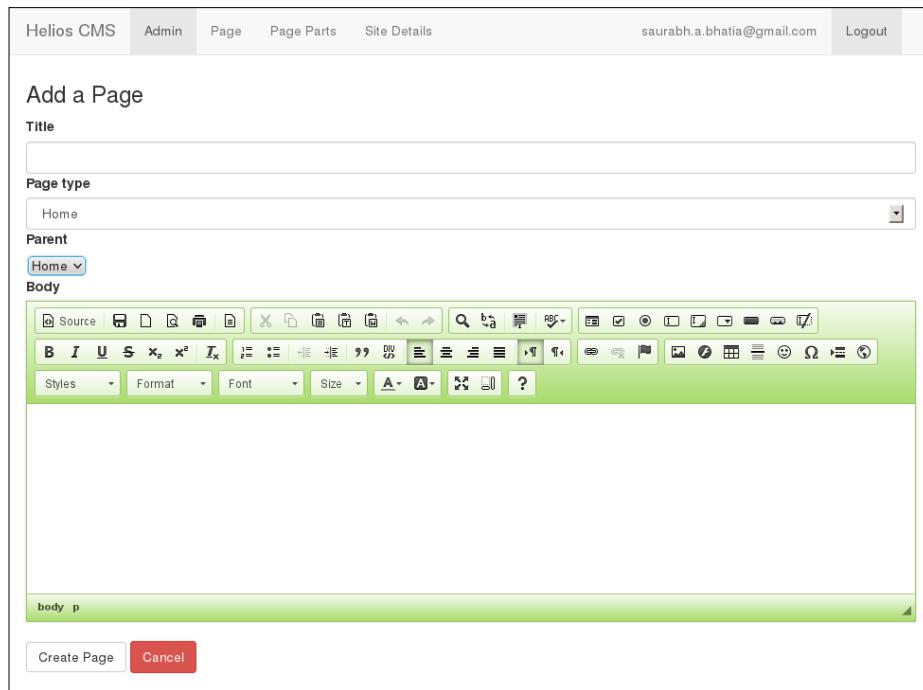
15. In our `page` model, we will add inverse association. This is to check for both parent and child and span the tree both ways:

```
has_many :child_page, :class_name => 'Page',
         :inverse_of => :parent_page
belongs_to :parent_page, :class_name => 'Page',
             :inverse_of => :child_page
```

16. We can now add a page to the form as a parent. Also, this method will create a tree structure and a parent-child relationship between the two pages:

```
app/views/pages/_form.html.haml
.field
  = f.label "Parent"
  %br/
  = f.collection_select(:parent_page_id, Page.all, :id,
                        :title, :class => "form-control")
.field
  = f.label :body
  %br/
  = f.cktext_area :body, :rows => 20, :ckeditor =>
    {:uiColor => "#AADC6E", :toolbar => "mini"}
  %br/
.actions
  = f.submit :class=>"btn btn-default"
  =link_to 'Cancel', pages_path, :class=>"btn btn-danger"
```

17. We can see the drop-down list with names of existing pages, as shown in the following screenshot:



18. Finally, we will display the parent page:

```
views/pages/_form.html.haml
.field
  = f.label "Parent"
  %br/
  = f.collection_select(:parent_page_id, Page.all, :id,
    :title, :class => "form-control")
```

19. In order to display the parent, we will call it using the association we created:

```
app/views/pages/index.html.haml
- @pages.each do |page|
  %tr
    %td= page.title
    %td= page.body.html_safe
    %td= page.parent_page.title if page.parent_page
```

Objective complete – mini debriefing

Mongoid is an ODM that provides an ActiveRecord type interface to access and use MongoDB. MongoDB is a document-oriented database, which follows a no-schema and dynamic-querying approach. In order to include Mongoid, we need to make sure we have the following module included in our model:

```
include Mongoid::Document
```

Mongoid does not rely on migrations such as ActiveRecord because we do not need to create tables but documents. It also comes with a very different set of datatypes. It does not have a datatype called text; it relies on the string datatype for all such interactions. Some of the different datatypes are as follows:

- ▶ **Regular expressions:** This can be used as a query string, and matching strings are returned as a result
- ▶ **Numbers:** This includes integer, big integer, and float
- ▶ **Arrays:** MongoDB allows the storage of arrays and hashes in a document field
- ▶ **Embedded documents:** This has the same datatype as the parent document

We also used Haml as our markup language for our views. The main goal of Haml is to provide a clean and readable markup. Not only that, Haml significantly reduces the effort of templating due to its approach.

In this task, we created a page model and a controller. We added a field called `page_type` to our page. In order to set a home page, we created a scope to find the documents with the page type `home`:

```
scope :home, ->{where(page_type: "Home")}
```

We then called this scope in our controller, and we also set a specific layout to our show page and home page. This is to separate the layout of our admin and pages.

The website structure can contain multiple levels of nesting, which means we could have a page structure like the following: **About Us | Team | Careers | Work Culture | Job Openings**

In the preceding structure, we were dealing with a page model to generate different pages. However, our CMS should know that **About Us** has a child page called **Careers** and in turn has another child page called **Work Culture**. In order to create a parent-child structure, we need to create a self-referential association. In order to achieve this, we created a new model that holds a reference on the same model page.

We first created an association in the page model with itself. The line `:inverse_of` allows us to trace back in case we need to span our tree according to the parent or child:

```
has_many :child_page, :class_name => 'Page', :inverse_of => :parent_page

belongs_to :parent_page, :class_name => 'Page', :inverse_of => :child_page
```

We created a page relationship to handle this relationship in order to map the parent ID and child ID. Again, we mapped it to the class page:

```
belongs_to :parent, :class_name => "Page"

belongs_to :child, :class_name => "Page"
```

This allowed us to directly find parent and child pages using associations.

In order to manage the content of the page, we added CKEditor, which provides a feature-rich toolbar to format the content of the page. We used the CKEditor gem and generated the configuration, including carrierwave. For carrierwave to work with mongoid, we need to add dependencies to Gemfile:

```
gem 'carrierwave', :github => "jnicklas/carrierwave"

gem 'carrierwave-mongoid', :require => 'carrierwave/mongoid'

gem 'mongoid-grid_fs', github: 'ahoward/mongoid-grid_fs'
```

MongoDB comes with its own filesystem called GridFs. When we extend carrierwave, we have an option of using a filesystem and GridFs, but the gem is required nonetheless. carrierwave and CKEditor are used to insert and manage pictures in the content wherever required.

We then added a route to mount the CKEditor as an engine in our routes file. Finally, we called it in a form:

```
= f.cktext_area :body, :rows => 20, :ckeditor => { :uiColor =>
 "#AADC6E", :toolbar => "mini"}
```

CKEditor generates and saves the content as HTML. Rails sanitizes HTML by default and hence our HTML is safe to be saved.

The admin page to manage the content of pages looks like the following screenshot:

Helios CMS		Admin	Page	saurabh.a.bhatia@gmail.com	Logout
Manage Pages					
Title	Body	Parent			
About	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis nec odio eget lorem congue imperdiet. Nam vulputate massa luctus molestie sagittis. Duis sagittis diam elit, egestas pharetra sem venenatis vitae. Vivamus vestibulum suscipit metus, nec porttitor lectus condimentum a. Praesent in ipsum dolor. Integer interdum rhoncus dolor eleifend fermentum. Sed sit amet tellus molestie, faucibus sapien et, ultrices nisi. Donec ultricies turpis venenatis ipsum faucibus tincidunt. In in felis dolor. Aenean sed tortor luctus sem varius facilisis. In pretium urna id pellentesque fermentum. Ut hendrerit imperdiet venenatis. Phasellus ultrices velit non metus	Home			

Managing page parts

This task deals with the creation and management of page parts. Page parts are snippets of code, which we will use to render in the page. These parts can be banners, YouTube video channels, photos, polls, and so on. We will create a model for page parts and this will effectively manage content for different parts of our page.

Engage thrusters

We will begin by adding page parts to our CMS system:

1. Generate the page parts model:

```
helioscms$ rails g model part title:string content:string  
meta:string part_type_id:string  
  invoke  mongoid  
  create   app/models/part.rb  
  invoke  test_unit  
  create   test/models/part_test.rb  
  create   test/fixtures/parts.yml
```

2. We will now generate the model for part_types:

```
:~/helioscmss$ rails g model part_type name:string  
  invoke  mongoid  
  create   app/models/part_type.rb  
  invoke  test_unit
```

```
create      test/models/part_type_test.rb
create      test/fixtures/part_types.yml
```

3. We will now associate the parts and part_types fields:

```
app/models/part_type.rb
class PartType
  include Mongoid::Document
  field :name, type: String

  has_many :parts

end

app/models/part.rb
class Part
  include Mongoid::Document
  field :title, type: String
  field :part_type_id, type: String
  field :content, type: String
  field :meta, type: String
  field :user_id, type: String

  belongs_to :page
  belongs_to :part_type

end
```

4. Let's add some parts by firing up the Rails console:

```
helioscms$ rails c
Loading development environment (Rails 4.0.0)
1.9.3-p327 :001 > part = Part.new
=> #<Part _id: a833e8207277751d1a000000, title: nil, part_type_
id: nil, content: nil, meta: nil, user_id: nil,>
1.9.3-p327 :002 > part.title = "YouTube Channel"
=> "YouTube Channel"
1.9.3-p327 :003 > part.save!
  MOPED: 127.0.0.1:27017 COMMAND      database=admin
command={:ismaster=>1} runtime: 3.5448ms
  MOPED: 127.0.0.1:27017 INSERT      database=project5_
development collection=parts documents=[{"_id"=>BSON::ObjectId('a8
33e8207277751d1a000000'), "title"=>"YouTube Channel"}] flags=[]
                                COMMAND      database=project5_
development command={:getlasterror=>1, :w=>1} runtime: 1.3837ms
=> true
```

5. We will now add part types to the part form so that we can save it during their creation:

```
app/views/parts/_form.html.haml
.field
  = f.label :part_type_id
  = f.select(:part_type_id, options_from_collection_for_
select(PartType.all, :id, :name), {:prompt => 'Please Choose'},
:klass => "form-control")
```

6. The following code shows what the full form looks like:

```
views/plans/_form.html.haml

= form_for @part do |f|
  - if @part.errors.any?
    #error_explanation
      %h2= "#{pluralize(@part.errors.count, "error")}" prohibited this
      part from being saved:
    %ul
      - @part.errors.full_messages.each do |msg|
        %li= msg

.field
  = f.label :title
  = f.text_field :title, :klass=>"form-control"
.field
  = f.label :part_type_id
  = f.select(:part_type_id, options_from_collection_for_
select(PartType.all, :id,
:name), {:prompt => 'Please Choose'}, :klass => "form-
control")
.field
  = f.label :content
  = f.cktext_area :content, :rows => 20, :ckeditor =>
  {:uiColor => "#AADC6E", :toolbar => "mini"}
.field
  = f.label :meta
  = f.text_field :meta, :klass=>"form-control"
.field
  = f.hidden_field :user_id, :value=>current_user.id
%br/
.actions
  = f.submit 'Save',:klass=>"btn btn-default"
  = link_to 'Cancel', parts_path,:klass=>"btn btn-
danger
```

7. In order to call the page_parts extension, we will use the association between page and page_parts.

8. To see this, we will make a call on page and then call the parts related to that page. As a result, you will see the following screenshot:

```
Loading development environment (Rails 4.1.0.beta1)
1.9.3p327 :001 > page = Page.first
  MOPED: 127.0.0.1:27017 COMMAND      database=admin command={:ismaster=>1} runtime: 1.5555ms
  MOPED: 127.0.0.1:27017 QUERY      database=project5 development collection=pages selector={"$query":{}, "$orderby":{:_id=>1}} flags
=1 limit=1 skip=0 batch_size=nil fields=nil runtime: 40.726ms
=> #<Page _id: 2c83b00e7277750e5e000000, title: "About", body: "<p style='text-align:justify'>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis ne odio eget lorem congue imperdiet. Nam vulputate massa luctus molestie sagittis. Duis sagittis diam elit, egesta s pharetra sem venenatis vitae. Vivamus vestibulum suscipit metus, nec porttitor lectus condimentum a. Praesent in ipsum dolor. Integer i nterdum rhoncus dolor eleifend fermentum. Sed sit amet tellus molestie, faucibus sapien et, ultrices nisi. Donec ultricies turpis venenat is ipsum faucibus tincidunt. In in felis dolor. Aenean sed tortor luctus sem varius facilisis. In pretium urna id pellentesque fermentum. Ut hendrerit imperdie venenatis. Phasellus ultrices velit non metus sagittis faubibus.</p>\r\n<p style='text-align:justify'>Curabitur consequat mauris quis vestibulum faubibus. Praesent quam eros, lacinia vitae molestie sed, lacinia vitae enim. Donec aliquet risus e get dignissim iaculis. Donec elementum quam leo, quis tempor nisi scelerisque quis. Morbi venenatis auctor dui, eget sollicitudin mauris blandit auctor. Nulla fringilla nibh id consectetur hendrerit. Vivamus congue, ipsum id pretium auctor, ipsum orci imperdiet libero, vita e consequat tellus turpis at erat. Donec vitae neque imperdiet, pellentesque mauris et, bibendum leo. In vulputate massa non erat fringil la accumsan. Nulla luctus ligula vel lorem sodales sollicitudin. Pellentesque habitant morbi tristique senectus et netus et malesuada fam es ac turpis egestas. Vestibulum ante ipsum primis in faubibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sed magna lacu s. Curabitur semper pharetra purus eu placerat. Fusce ut hendrerit est.</p>\r\n", is_home: "0", page_type: "Home", part_ids: nil, parent_
page_id: BSON::ObjectId('0beb07d0727775408a000000'), user_id: nil>
1.9.3p327 :002 > page.parts
  MOPED: 127.0.0.1:27017 QUERY      database=project5 development collection=parts selector={"page_id":>BSON::ObjectId('2c83b00e7277750e5e000000')} flags=1 limit=0 skip=0 batch_size=nil fields=nil runtime: 18.165ms
=> [#<Part _id: 3b84d920727775139b000000, title: "Bulletin", part_type_id: "2870fa16727775100e020000", content: "<p>Bulletin Board</p>\r\n", meta: {"announcements": true}, user_id: "4fe2da877277752047000000", page_id: BSON::ObjectId('2c83b00e7277750e5e000000'), #<Part _id: a833e8207277751dia000000, title: "YouTube Channel", part_type_id: nil, content: nil, meta: nil, user_id: nil, page_id: BSON::ObjectId('2c83b00e7277750e5e000000')>]
1.9.3p327 :003 > 
```

Objective complete – mini debriefing

We created a page parts model in this task. We also created page part types in order to classify and arrange them. We have also created an association between page and page parts. Hence, we can now assign a single page part to multiple pages. Also, we can see all the parts associated with a page.

At the end of this task, our page part creation page should look like the following screenshot:

The screenshot shows the 'Add part' interface in the Helios CMS. At the top, there's a navigation bar with 'Helios CMS', 'Admin', 'Page', 'Page Parts', the user email 'saurabh.a.bhatia@gmail.com', and a 'Logout' button. Below the navigation is a title 'Add part'. The main area has three sections: 'Title' with a text input field, 'Part type' with a dropdown menu showing 'Please Choose', and 'Content' which contains a WYSIWYG editor with a toolbar. The toolbar includes icons for Source, Bold, Italic, Underline, etc. Below the toolbar is a larger text area with placeholder text 'body p'. At the bottom of the form are 'Save' and 'Cancel' buttons.

Creating a Haml- and Sass-based template

Now that we have the content defined for our pages, we will start building the frontend. We will keep the frontend as simple as possible and just render the information we have created using our CMS. This task deals with the creation of the frontend and how to separate it from the backend.

Engage thrusters

Let's get started with the process of frontend creation:

1. Inside your `app/assets` folder, create a file called `front.css`. In Rails we have an advantage of asset pipeline. We can use this to separate the frontend assets. We will create a manifest file called `front.css` and define front end-related stylesheets under it:

```
app/assets/stylesheets/front.css
/*
 * This is a manifest file that'll be compiled into front.css,
which will include all the files
 * listed below.
 *
 * indicates any CSS and SCSS file within the lib/assets/
stylesheets/front_end, vendor/assets/stylesheets/front_end
directory.
 * or vendor/assets/stylesheets of plugins, if any, can be
referenced here using a relative path.
 *
 * You're free to add application-wide styles to this file and
they'll appear at the top of the
 * compiled file, but it's generally better to create a new file
per style scope.
 *
 *= require_self
 */
```

2. We will also place these files under the folder called `front_end` and create a blank SCSS file under it:

```
helioscms/app/assets/stylesheets$ mkdir front_end
helioscms/app/assets/stylesheets/front_end$ touch structure.scss
```

3. We will now load the `structure.scss` file from our manifest file:

```
app/assets/stylesheets/front.css
/*
 * This is a manifest file that'll be compiled into front.css,
which will include all the files
 * listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/
stylesheets, vendor/assets/stylesheets,
 * or vendor/assets/stylesheets of plugins, if any, can be
referenced here using a relative path.
 *
 * You're free to add application-wide styles to this file and
they'll appear at the top of the
 * compiled file, but it's generally better to create a new file
per style scope.
 *
 *= require_self
 *= require font-awesome
 *= require front_end/structure

*/
```

4. We will follow the same procedure to create a `front.js` manifest file in `assets/javascripts`.
5. Let's quickly define a simple three column layout with a header, footer, and a container. We will do this inside our `structure.scss` file:

```
app/assets/stylesheets/front_end/structure.scss
#primary, #content, #secondary {
    height: 300px;
    padding: 50px 0;
}

#container {
    width: 1200px;
    margin: 0 auto;
}

#primary {
    float: left;
    width: 150px;
    background: #eee;
    padding-right: 10px;
    padding-left: 10px;
```

```
}

#content {
    float: left;
    width: 800px;
    height: 300px;
    background: #ccc;
    padding-left: 10px;
    padding-right: 10px;
}

#secondary {
    float: left;
    width: 150px;
    background: #ddd;
    padding-left: 10px;
    padding-right: 10px;
}

#footer {
    clear: both;
    padding-top: 3px;
}

#header {
    background: #fff;
    height: 100px;
}
```

6. We will also add a basic horizontal menu to the application:

```
app/assets/stylesheets/front_end/structure.scss
#menu ul
{
    margin: 0px;
    padding: 0px;
    list-style-type: none;
}

#menu a
{
    display: block;
    width: 8em;
    color: white;
    background-color: #000099;
    text-decoration: none;
```

```
    text-align: center;
}

#menu a:hover
{
    background-color: #6666AA;
}
#menu li
{
    float: left;
    margin-right: 0.5em;
}
```

7. Our front end layout is currently different. We will make a call on the manifest CSS and JS files in our header and define sections in the page. Also, we should change our extension's layout to Haml because the first layout created is `html.erb` by default:

```
app/views/layouts/page_layout.html.haml
!!!
%html
    %head
        %title
            = stylesheet_link_tag "front"
            = javascript_include_tag "front"
            = csrf_meta_tags
    %body
        #container
            #header

        #menu

        #primary
            %p Primary Sidebar
            #content

        #secondary
            %p Secondary Sidebar
            #footer
```

8. Finally, we will add some fonts before we start rendering the content into our page. We will use Google Fonts for simple usage:

```
views/layouts/page_layout.html.haml
  %link{href: "http://fonts.googleapis.com/
css?family=Cherry+Swash", rel: "stylesheet", type: "text/css"}/
  %link{href: "http://fonts.googleapis.com/css?family=Flamenco",
rel: "stylesheet", type: "text/css"}/
```

9. We need to apply this layout only to the limited actions in our controller:

```
app/controllers/pages_controller.rb
```

```
layout 'page_layout', only: [:home_page, :show]
```

Objective complete – mini debriefing

In this task, we concentrated on creating a frontend and separated it completely from the backend in terms of look and feel. The advantage of this approach is that we can use any CSS and JS framework we want in the frontend without interfering with the backend. Zurb Foundation, Bootstrap, Less, or any other HTML5 and CSS3 framework can be used for the frontend.

```
= stylesheet_link_tag "front" = javascript_include_tag "front"
= javascript_include_tag "front"
```

Finally, we had to apply this layout to some select actions in our controller. This is because our page controller's actions—index, new, edit, create, and update—are administrator's actions. The actions show and home_page are supposed to display the page. Hence, the layout is applied to only these actions:

```
layout 'page_layout', only: [:home_page, :show]
```

The final output of our work in creating the frontend page is as seen in the following screenshot:



Generating the content and pages

We have already created the backend and also set the base for the frontend. However, we need to start rendering the content in the front end. We also want a dynamically generated menu from the pages we have created. We want the backend to play well with the front end page we just created. In this task, we will add site-related information that renders all the content on the front end page.

Engage thrusters

The following steps are used to render the content and also add some general site details:

1. We will first create a scaffold for the site details:

```
$ rails g scaffold site_detail title:string organization:string  
address:string facebook:string twitter:string google_  
plus:string skype:string linkedin:string google_analytics:string  
telephone:string  
invoke  mongoid  
create    app/models/site_detail.rb  
invoke    test_unit  
create    test/models/site_detail_test.rb  
create    test/fixtures/site_details.yml  
invoke    resource_route  
route     resources :site_details  
invoke    inherited_resources_controller  
        create    app/controllers/site_details_controller.rb  
invoke    erb  
create    app/views/site_details  
create    app/views/site_details/index.html.erb  
create    app/views/site_details/edit.html.erb  
create    app/views/site_details/show.html.erb  
create    app/views/site_details/new.html.erb  
create    app/views/site_details/_form.html.erb  
invoke    test_unit  
        create    test/controllers/site_details_controller_test.rb  
invoke    helper  
create    app/helpers/site_details_helper.rb  
invoke    test_unit  
        create    test/helpers/site_details_helper_test.rb  
invoke    jbuilder
```

```
create      app/views/site_details/index.json.jbuilder
create      app/views/site_details/show.json.jbuilder
invoke assets
invoke coffee
  create    app/assets/javascripts/site_details.js.coffee
invoke scss
  create    app/assets/stylesheets/site_details.css.scss
invoke scss
  identical app/assets/stylesheets/scaffolds.css.scss
```

2. Be sure to remove the scaffolds.css.scss file, otherwise it will conflict with our default CSS.

3. First generate the carrierwave uploader and call the uploaded file:

```
helioscms$ rails g uploader file
```

4. We will then add a field to the SiteDetail model:

```
app/models/site_detail.rb
class SiteDetail
  include Mongoid::Document
  field :title, type: String
  field :organization, type: String
  field :address, type: String
  field :facebook, type: String
  field :twitter, type: String
  field :google_plus, type: String
  field :skype, type: String
  field :linkedin, type: String
  field :google_analytics, type: String
  field :telephone, type: String

  mount_uploader :logo, FileUploader
end
```

5. The form to save the site details looks as follows:

The screenshot shows a web-based form titled "Add Details for your Site" within a "Helios CMS" interface. The form fields include:

- Title:** Helios Tech
- Organization:** Helios Tech LLC
- Address:** Suit 10, 23rd Street,Broadway, NYC
- Facebook:** facebook.com/HeliosTech
- Twitter:** twitter.com/heliostech
- Google plus:** https://plus.google.com/u/0/108146960744308392533
- Skype:** heliostech
- LinkedIn:** http://www.linkedin.com/company/738023
- Google analytics:** A code editor containing the following JavaScript snippet:

```
<script type="text/javascript">
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-31041958-1']);
_gaq.push(['_trackPageview']);

(function() {
  var ga = document.createElement('script'); ga.type = 'text/javascript'; ga.async = true;
  ga.src = ('https:' == document.location.protocol ? 'https://ssl' : 'http://www') + '.google-
  analytics.com/ua';
  document.documentElement.appendChild(ga);
})();

```
- Telephone:** (This field is empty)

At the bottom of the form are two buttons: "Save" (in white) and "Cancel" (in red).

6. Now, in order to display these values in our site frontend, we will first make a call on SiteDetail and Page. We will call the SiteDetail and page value:

app/controllers/pages_controller.rb

```
before_action :set_site, only: [:home_page, :show]
layout 'page_layout', only: [:home_page, :show]
```

```
def home_page
  @page = Page.find_by(page_type: "Home") rescue nil
  @pages = Page.all

end

# GET /pages/1
# GET /pages/1.json
def show

  @pages = Page.all

end

private
def set_site

  @site = SiteDetail.first

end
```

7. We will then add these values to the page. First add the `page` and `site` value in the title bar:

```
"app/" before the path "views/layouts/page_layout.html.haml"
views/layouts/page_layout.html.haml
!!!
%html
  %head
    %title
      = @page.title
      | #{@site.title}
```

8. We will then add site values to the footer, site title, and logo to the header.
9. We will also call all the pages and loop them in line to generate our page navigation. We will lastly call the body inside the content tag so that the content is rendered there:

```
app/views/layouts/page_layout.html.haml
%body
  #container
  #header
    %p= image_tag @site.logo_url.to_s,
      :alt=>"#{@site.title}"
  #menu
  %ul
    - @pages.each do |page|
      %li= link_to page.title, page
    #primary
    %p Primary Sidebar
```

```
#content
%p= @page.body.html_safe
#secondary


Secondary Sidebar


#footer


### = @site.organization



%p
= link_to '<i class="fa fa-facebook"></i>'.html_safe,
@site.facebook, :target=>"blank"
| #{link_to '<i class="fa fa-twitter"></i>'.html_safe,
@site.twitter, :target=>"blank"} | #{link_to '<i
class="fa fa-linkedin"></i>'.html_safe,
@site.linkedin, :target=>"blank"} | #{link_to '<i
class="fa fa-skype"></i>'.html_safe, @site.skype,
:target=>"blank"}


%p
= @site.address
%br/
= @site.telephone
```

Objective complete – mini debriefing

At the end of this task, we created a model for storing the site details. In our controller, we called the site details and assigned the instance variable to actions where we need our site object:

```
before_action :set_site, only: [:home_page, :show]

private
def set_site
  @site = SiteDetail.first
end
```

We called the values of site title, address, and contact details in the footer, and content in the content tag. The advantage of using Haml and Sass is clean markup, with very good indentation and code readability. Sass is like an extension of CSS, which compiles to CSS code. One of the main advantages of using Sass is the usage of a variable to make some of the code reusable. Values such as font sizes, colors, and font-family can easily be made dry using Sass variables. We can do a quick refactor of our Sass using a variable for defining the font-family as follows:

```
app/assets/stylesheets/front_end/structure.scss
$primary-font: 'Cherry Swash', cursive;

#footer {
```

```
clear: both;  
padding-top: 3px;  
font-family: $primary-font;  
}  
  
#header {  
background: #fff;  
height: 100px;  
font-family: $primary-font;  
}
```

The other option to keep the CSS code clean is using Less CSS (<http://lesscss.org/>). This extends the CSS to use features such as functions and mixins too. We can see in the following screenshot how Sass is compiled into and is rendered with the site details also displayed:



Implementing asset caching

In our CMS there are several kinds of assets. As we build themes we will beautify them with varied JavaScript, CSS, and images. In order to keep the speed of our sites fast in the frontend, we will use asset caching in Rails.

Engage thrusters

The steps to follow will be to cache the content and speed up our site, as follows:

1. We will first make sure we have the right asset-related gems in Gemfile:

```
gem 'sass-rails', '~> 4.0.0'  
gem 'uglifier', '>= 1.3.0'  
gem 'coffee-rails', '~> 4.0.0'
```

This will enable all kinds of assets and make it ready for production.

2. We will first enable asset compression inside our production.rb file:

```
config/environments/production.rb  
config.assets.compress = true
```

3. We will continue to edit the same file:

```
config.assets.compress = true  
# Compress JavaScripts and CSS.  
config.assets.js_compressor = :uglifier  
config.assets.css_compressor = :sass
```

4. We will now fix the asset folder to tmp/cache/assets:

```
config/environments/production.rb  
config.assets.cache =  
  ActiveSupport::Cache::FileStore.new("tmp/cache/assets")
```

5. We need to make sure that we run the following step before deployment:

```
helioscms$ rake assets:precompile
```

6. In order to make use of memcached in our application, we will add a gem called dalli. The dalli gem is a replacement of the memcache client:

```
gem 'dalli'
```

7. We will configure the cache in our production.rb file:

```
config/environments/production.rb  
config.cache_store = :dalli_store  
config.action_controller.perform_caching = true
```

8. We will also add a simple action cache so that we always cache the layout along with the cache. In our pages_controller.rb file, we will add an action caching method:

```
app/controllers/pages_controller.rb  
def home_page  
  expires_in 5.minutes
```

```
sleep 15

@page = Page.home.first
cache_client = Dalli::Client.new('localhost:11211')

@pages = Page.all
end
```

9. Lastly, we will initiate a cache client and store a cached object in it:

```
app/controllers/pages_controller.rb
@page = Page.find_by(page_type: "Home")
cache_client = Dalli::Client.new('localhost:11211')

cache_client.set('Home', @page)
value = cache_client.get('Home')
```

10. So, finally our method for home page looks as follows:

```
def home_page
  expires_in 5.minutes
  sleep 15

  @page = Page.home.first
  cache_client = Dalli::Client.new('localhost:11211')

  cache_client.set('Home', @page)
  value = cache_client.get('Home')

  @pages = Page.all
end
```

Objective complete – mini debriefing

This is an extremely basic caching technique that comes as an extension to Rails. We added memcached for caching the page beforehand. This will help us to speed up our site's frontend. We looked at how to enable memcached for the application using the `dalli` gem. Memcached is a distributed key-value store for storing memory objects like objects, sessions, strings, API, and data calls. In a way it's a technique to store and retrieve temporary data. This data is stored in the form of an array and is quickly retrieved as soon as the page loads, instead of going back to the database and calling the page again. This avoids unnecessary queries and thus reduces the database load. This technique also saves API calls because for applications such as Twitter clients, the number of requests is a very important criterion. Hence, we first installed memcached on our local system. We then enabled `Dalli` in our `production.rb` file:

```
Dalli::Client.new('localhost:11211')
```

This will directly access the memcached at its port number and initiate a new client object. We first defined the time for cache expiry:

```
expires_in 5.minutes
```

In order to keep the transaction fast, the application pre-fills the cache with a value. This, however, can lead to another problem. It is quite possible that the same key is being accessed by multiple clients. Also, if the cache is empty, it could be filled with multiple keys. Hence, memcached gives an option called `sleep`, which provides a lock for the time defined in `sleep`:

```
sleep 15
```

If two processes are accessing the same key at the same time, then `sleep` will tell the other process that the cache is empty, while waiting for the sleep time to finish. Once done, the lock is released and autoassigned to the next value in the queue. In order to store a value in memcached, we used the following `set` method:

```
cache_client.set('Home', @page)
```

The `set` method includes the key ('`Home`') and the value (`@page`). For retrieving the value of the page, we used a simple `get` method:

```
value = cache_client.get('Home')
```

This value is retrieved using a key called `Home`. We must note that action caching is deprecated in Rails 4. We will have to use a third-party caching technique such as memcached to perform action caching. Fragment caching is another strategy where we cache certain parts of a page instead of the entire page, which is also a very commonly used technique and works nicely out of the box in Rails. We will cover this in our later projects.

Mission accomplished

We laid down the foundation for a dynamic CMS in this project. We built a backend admin with a functionality that could create pages and parts. We also created a frontend and did the markup using Haml and SCSS. SCSS is fast to load and easy to manage. It also fits neatly in the Rails asset pipeline. Hence, it is a recommended form of markup with Rails. Some of the ideas we looked at in this project were as follows:

- ▶ We replaced ActiveRecord with Mongoid for our database and model
- ▶ We created an admin area and made devise function only for the admin
- ▶ We saw how namespaces in controllers and routes work
- ▶ We created a self referential association, a parent and a child association on the same model

- ▶ We integrated CKEditor with our application
- ▶ We then created different layouts and manifests for our frontend and backend
- ▶ We looked at how Haml and Sass markup are done and their probable advantages
- ▶ Lastly, we looked at a memcached-based caching strategy for caching our actions

Hotshot challenges

We need to take our CMS to the next level. The exercise contains a few ideas worth trying out:

- ▶ Using nested attributes assign parts to a page.
- ▶ Adding a responsive HTML5 layout to the frontend.
- ▶ Adding validation for the home page. There should always be only one page called home.
- ▶ Adding tests to test content created with CKEditor.

Project 6

Creating an Analytics Dashboard using Rails and Mongoid

We rely a lot on various analytics tools in our web applications. Google Analytics, Mixpanel, Kissmetrics, and Crazy Egg are some of the most popular web-analytics tools that give a deep insight into who's visiting the website, from where, and what pages are getting the most hits. These analytics help in addressing demographic-based issues, improving the user experience on the site.

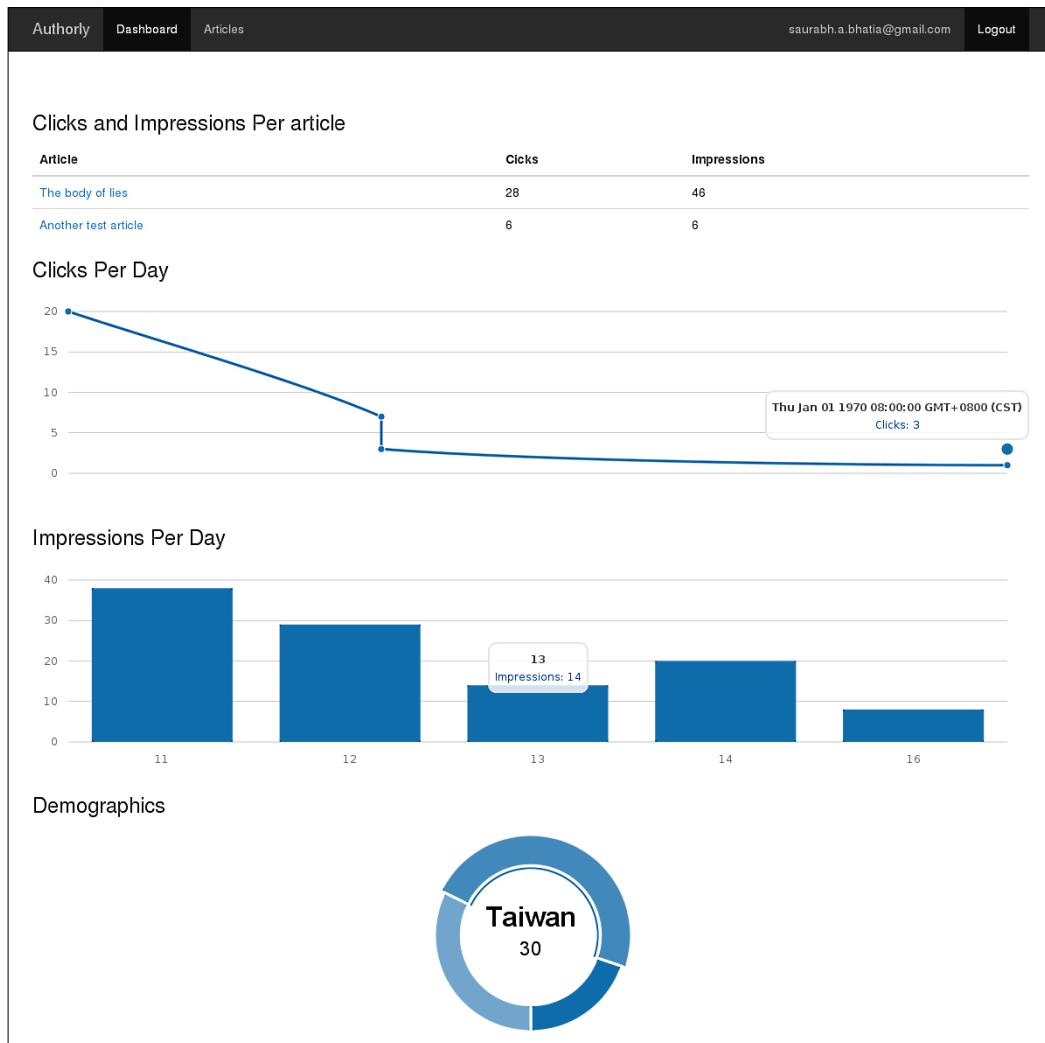
Mission briefing

In this project, we will create an analytics dashboard, which will give the user an insight on which kind of content is getting what kind of traffic. There are three types of behavior that we will track with our application:

- ▶ Clicks
- ▶ Views
- ▶ Visits

Clicks and views will be tracked for the users who have logged in. Visits are for the users who unknown and are not logged in. We will use **MongoDB** to track and store this data. Also, we will create charts of different types in order to visualize our data. MongoDB is scalable and is meant to be fault tolerant.

We will name our application Authorly and the following is a glimpse of what we are going to achieve:



Why is it awesome?

Sometimes analytics and visibility for our data needs to be part of our system. Also, if this dashboard is easy to roll out and manage, you can build an entire highly customized system in the long term. This data is valuable for the administrators of the system. In our application, we will create articles and give users the flexibility to track clicks and views to their articles through a dashboard.

Analytics comprises the following three tasks:

- ▶ Collecting the data
- ▶ Analyzing the collected data
- ▶ Reporting the data

At the end of this project, we will be able to build a fully functional analytics dashboard.

Your Hotshot objectives

While building this application, we will go through the following tasks:

- ▶ Creating a MongoDB database
- ▶ Creating a click-tracking mechanism
- ▶ Creating a visit-tracking mechanism
- ▶ Writing map-reduce and aggregation to fetch and analyze the data
- ▶ Creating a dashboard to display clicks and impression values
- ▶ Creating a line graph of the daily clicking activity
- ▶ Creating a bar graph of the daily visit activity
- ▶ Creating a demographic-based donut chart

Mission checklist

We need the following software installed on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MongoDB
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Devise
- ▶ morris.js for charts
- ▶ Git
- ▶ A tool for mockups
- ▶ jQuery
- ▶ ImageMagick and RMagick

Creating a MongoDB database

In this task, we will work towards setting up the base for our application. This includes setting up mongoid, rolify, and creating articles. This task is more like a revision of some of the concepts that we have covered in the book already. The new thing here is that we are doing it all with Mongoid.

Prepare for lift off

In order to start working on this project, we will first have to add the `mongoid` gem to the `Gemfile`:

```
Gemfile
gem 'mongoid', github: 'mongoid/mongoid'
```

Bundle the application and run the `mongoid` generator:

```
$ rails g mongoid:config
```

At the time of writing this book, the master branch of rolify is compatible only with the master branch of mongoid. So, in order to ensure that both work well together, we need to keep both our Mongoid and rolify on the master branch.

Engage thrusters

The steps for creating a MongoDB database are as follows:

1. We will take the first step in this task by setting up the skeleton of the application.
2. We will install rolify from the master branch by adding it to the `Gemfile` and run `bundle`:

```
Gemfile
gem 'rolify', :github => 'EppO/rolify'
authorly $bundle install
```

3. We will then generate the configuration file for rolify:

```
authorly$rails g rolify Role User -o mongoid
      invoke  mongoid
      create   app/models/role.rb
      invoke  test_unit
      create   test/models/role_test.rb
      create   test/fixtures/roles.yml
      insert   app/models/role.rb
      insert   app/models/user.rb
      create   config/initializers/rolify.rb
```

4. The initializers generated in order to access mongoid instead of ActiveRecord looks like the following code:

```
config/initializers/rolify.rb
```

```
Rolify.configure do |config|
  config.use_mongoid
end
```

5. We will generate an article's model, view, and controller. This will allow the users to create articles:

```
authorly$ rails g scaffold article title:string body:text
```

6. MongoDB generates pretty ugly URLs, with 12-byte long **Binary JSON (BSON)** type IDs trailing them. We need to create good looking URLs with MongoDB. For this, we will use the `mongoid_slug` gem with our application. Again, here we are using the master branch of GitHub to maintain the compatibility with Rails 4 and mongoid 4 beta versions:

```
Gemfile
```

```
gem 'mongoid_slug', github: 'digitalplaywright/mongoid-slug'
```

7. After adding it to `Gemfile`, run `bundle install`.
8. In order to set up the slugging mechanism, we will first include the `Mongoid::Slug` module in our `article` model:

```
app/models/article.rb
```

```
class Article
  include Mongoid::Document
  include Mongoid::Slug

  field :title, type: String
  field :body, type: String
  field :user_id, type: String
  belongs_to :user
end
```

9. Also, we need to store the history of our URL slugs to avoid 404 errors in case the slug changes. This will be stored in an array inside the `_slug` field in the `article` model:

```
app/models/article.rb
```

```
class Article
  include Mongoid::Document
```

```
include Mongoid::Slug

field :title, type: String
field :body, type: String
field :_slugs, type: Array, default: []
field :user_id, type: String
slug :title, :history => true
end
```

10. We will set up an article list such that it can be viewed by anyone without logging in as well as by people who are logged in. Before this step, please make devise is installed on your system:

```
app/controllers/articles_controller.rb

before_filter :authenticate_user!, except: [:show, :index]
def index

  @articles = Article.all

end
```

11. Lastly, do not forget to add a slug and user ID to the permitted parameters in your articles_controller file:

```
app/controllers/articles_controller.rb

private
  # Use callbacks to share common setup or constraints between
actions.
  def set_article
    @article ||= Article.find(params[:id])
  end

  # Never trust parameters from the scary internet, only allow
the white list through.
  def article_params
    params.require(:article).permit(:title, :body, :_slugs,
:user_id)
  end
```

Objective complete – mini debriefing

In this task, we started by assuming that devise and cancan have already been installed, as there was no change needed for any of them to work with mongoid. We directly proceeded to the step where we installed rolify with mongoid. We created a model for articles and restricted the access for the show and index pages being accessed by anyone. We then saw the use of a mongoid slug, a library that is used to create pretty and search-friendly URLs.

A good solution for slugs not only makes the URL pretty and search friendly, but also maintains a history of the changes done to the URLs. There are chances that the slug might change as it is dependent on the article's title. If a user edits the title, the slug is bound to change. However, if the article is popular and is used by several people, they might have bookmarked it. We used the history feature to maintain both old as well as new URLs, thus avoiding the 404 (URL not found) errors. We also added _slugs to the parameter's whitelist.

Creating a click-tracking mechanism

There is a difference between tracking clicks and tracking impressions. Clicks can be the traffic that is received through an organic search via search engines such as Google, or via searching the website, or whenever a click action is performed. Impression, on the other hand, is how many times the page has been viewed. It is possible that someone has bookmarked the page and repeatedly read an article. In this case, the act will be the counting of impressions. In our application, both clicks and impressions will be bound to the show method because that's what is mainly required to render the page.

Engage thrusters

We will now go ahead and create a click-tracking mechanism for our articles:

1. We will first create a model for clicks and associate it with the article:

```
app/models/click.rb
```

```
class Click
  include Mongoid::Document
  field :ip, type: String
  field :url, type: String
  field :article_id, type: String
  field :user_id, type: String

  belongs_to :article
end
```

2. In our article, we will associate our article model with the clicks too:

```
app/models/article.rb
```

```
has_many :clicks
```

3. We will first add methods to get the full path of the URL and get the IP address of the user clicking in our `show` method, inside our `articles_controller` file:

```
app/controllers/articles_controller.rb
def show
  @url = request.fullpath.to_s
  @ip = request.remote_ip
end
```

4. Now, we will track the click action whenever it is performed and the `show` method is fired. Also, we will save `article_id` with our click. We will do this with the following code:

```
app/models/concerns/record_data.rb
module RecordData
  extend ActiveSupport::Concern
  included do
    def self.record(url, ip, article_id, user_id)
      self.create!(url: url, ip: ip, article_id: article_id, user_id: user_id)
    end
  end
end

app/controllers/articles_controller.rb
def show
  @clicks = @article.track_clicks_per_article
  url = request.fullpath.to_s
  ip = request.remote_ip
  if user_signed_in? && (current_user.id != @article.user_id)
    Click.record(url, ip, @article.id, current_user.id.to_s)
  elsif !user_signed_in?
    Click.record(url, ip, country, city, @article.id, "anonymous")
  end
end
```

5. Now, we will have the click recorded every time a user clicks on the `show` method. For an anonymous user, the query looks like the following code:

```
MOPED: 127.0.0.1:27017 INSERT
database=project6_development collection=clicks
documents=[{"_id"=>BSON::ObjectId
('528243f37277750cd90a0000'), "url"=>"/articles/
the-body-of-lies", "ip"=>"127.0.0.1",
"article_id"=>BSON::ObjectId('528011687277750d4a000000'),
"user_id"=>"anonymous"}] flags=[]
```

6. For a logged-in user, the query looks like the following code:

```
MOPED: 127.0.0.1:27017 INSERT
database=project6_development collection=clicks
documents=[{"_id":>BSON::ObjectId
('5283648d7277750b6a050000'), "url":>"articles/the-body-of-lies", "ip":>"127.0.0.1",
"article_id":>BSON::ObjectId('528011687277750d4a000000') ,
"user_id":>"527ce7927277750d00000000"}] flags=[]
```

Objective complete - mini debriefing

In the preceding task, we created a simple click-tracking mechanism that executes and saves every time the show link is clicked in the frontend. We saved the ID of the article along with our click in order to see which article gets how many clicks:

```
@url = request.fullpath.to_s
@ip = request.remote_ip
```

We created a model concern to create a new click record every time the user clicks on the show action. In our previous project (*Project 4, Creating a Restaurant Menu Builder*), we created a controller concern for the subdomain. Here, we created a reusable class method that we can call on different models if we have to create a scorecard with those attributes. In order to include the class method in our model, we just included the module in our model and called the class method on our Click model. We also took measures to track the ID of the user if they are logged in. If the user is anonymous, we will know that the traffic is from a source where the user is not logged in. This will give us wholesome statistics on the clicks received on the article.

Creating a visit-tracking mechanism

In order to track visits and impressions, we will take a slightly different approach. We will use a gem called `impressionist` to track the page impressions. At the end of the task, we will also debate whether the solution is scalable or not. The difference between impressions and clicks lies in how the article is accessed. So, for example, if a user writes an article that is linked in another website and someone clicks on the link, this would count as a click. However, if a link is bookmarked and the user tries to access it from the bookmarks, it would count as an impression. Hence, we have tied both impressions and clicks to the show method.

Engage thrusters

We will now create view tracking for our articles:

1. We will first add the `impressionist` gem to our `Gemfile` and run `bundle`. Even here, we will keep our gem to master head so that we grab the latest version that is compatible with Rails 4 and mongoid 4:

```
gem 'impressionist', github: 'charlotte-ruby/impressionist'
```

2. We will now generate the `impressionist` initializer:

```
:~/authorly$ rails g impressionist --orm mongoid
      invoke  mongoid
      create   config/initializers/impression.rb
```

3. The `is_impressionable` method in the `article` model will allow `impressionist` to access the `article` mode:

```
app/models/article.rb
class Article
  include Mongoid::Document
  include Mongoid::Slug

  field :title, type: String
  field :body, type: String
  field :_slugs, type: Array, default: []
  field :user_id, type: String

  is_impressionable

  slug :title, :history => true
  belongs_to :user
  has_many :clicks
end
```

4. After associating with the model, we will have to pass the `article` object to `impressionist`:

```
app/controllers/articles_controller.rb
def show
  impressionist(@article, message: "A User has viewed your article")

  url = request.fullpath.to_s
  ip = request.remote_ip

  if user_signed_in? && (current_user.id != @article.user_id)
    Click.record(url, ip, @article.id, current_user.id.to_s)
  elsif !user_signed_in?
```

```
    Click.record(url, ip, @article.id, "anonymous")
  end
end
```

5. Also, we can set a filter to run `impressionist` for specific actions:

```
app/controllers/articles_controller.rb
class ArticlesController < ApplicationController
  before_action :set_article, only: [:show, :edit, :update, :destroy]
  before_filter :authenticate_user!, except: [:show, :index]
  impressionist :actions=>[:show]
end
```

6. We are now ready to track the page views. We, however, do not have a collection for the impressions yet. So, we will generate a model for `impression`:

```
authorly$ rails g model page_impression
impressionable_type:string impressionable_id:string
user_id:string controller_name:string action_name:string
view_name:string request_hash:string ip_address:string
session_hash:string message:string referrer:string
```

7. The impression model should also include the timestamps with it:

```
app/models/page_impression.rb
class PageImpression
  include Mongoid::Document
  include Mongoid::Timestamps::Created

  field :impressionable_type, :type => String
  field :impressionable_id, :type => String
  field :user_id, :type => String
  field :controller_name, :type => String
  field :action_name, :type => String
  field :view_name, :type => String
  field :request_hash, :type => String
  field :ip_address, :type => String
  field :session_hash, :type => String
  field :message, :type => String
  field :referrer, :type => String
end
```

8. We just need to ensure that the model is being saved properly. So, we will navigate to the `show` method to see the queries:

```
Processing by ArticlesController#show as HTML
Parameters: {"id"=>"the-body-of-lies"}
MOPED: 127.0.0.1:27017 QUERY      database=project6
development collection=articles selector={"_slugs"=>{"$in"=>["the-
body-of-lies"]}} flags=[] limit=1 skip=0 batch_size=nil fields=nil
```

```
runtime: 0.8295ms
    MOPED: 127.0.0.1:27017 QUERY      database=project6_
development collection=users selector={"$query"=>{"_id"=>BSON::ObjectId('527ce7927277750d00000000')}, "$orderby"=>{:_id=>1}}
flags=[] limit=-1 skip=0 batch_size=nil fields=nil runtime:
0.5881ms
    MOPED: 127.0.0.1:27017 INSERT      database=project6_
development collection=impressions documents=[{"_id"=>BSON::ObjectId('5283648d7277750b6a030000'), "impressionable_type"=>"Article",
'impressionable_id"=>"the-body-of-lies", "controller_name"=>"articles", "action_name"=>"show", "user_id"=>BSON::ObjectId('527ce7927277750d00000000'), "request_hash"=>"871961ef69818fd7f9e0bef510f583fd387144ef4e919ed132982144e930f8a", "session_hash"=>"457126f191ff2b6da6d92c9f6ceaa62f", "ip_address"=>"127.0.0.1", "referrer"=>"http://localhost:3000/articles", "updated_at"=>2013-11-13 11:37:49 UTC, "created_at"=>2013-11-13 11:37:49 UTC}]
flags=[]
    COMMAND      database=project6_
development command={:getlasterror=>1, :w=>1} runtime: 0.7574ms
```

9. In order to display the impressions, we just need to make a call to the `impressionist_count` method on the `article` object:

```
app/views/articles/show.html.erb
<%= "#{@article.impressionist_count} views so far!" %>
```

Objective complete – mini debriefing

This task included setting up the `impressionist` gem and associating it with the model and object. We generated an initializer to associate it with mongoid. In our controller, we added the `impressionist` method to record the impressions. We also added a `page_impression` model in order to save the impression-related data. The `impressionist` method, however, is not the best and the most scalable solution. The reason for this is every time the method counts, it starts counting from the beginning. With a large recordset of 8 to 10 million records to count, it would take more than 10 seconds just to fetch the count. A good way to save and count from our previous saved values is to use the `ensureIndex` option in MongoDB:

```
db.collection.ensureIndex
```

In order to add this to our model, we used the `index` method in mongoid. This method fires `ensureIndex` in MongoDB:

```
index ({impressionable_type: 1, impressionable_id: 1 ,user_id: 1, controller_name: 1, action_name: 1, view_name: 1, request_hash: 1, ip_address: 1, session_hash: 1, referrer: 1, message: 1}, {
name: "page_impression_index" })
```

If there are multiple fields to index, make sure to add a name to the index. This will keep the last counted value indexed and run the `impressionist` query after the value is indexed. This will also bring up the performance and decrease the `count` query time to less than 1 second.

Also, the default model generated does not have dates in it by default. In order to add `created_at` and `updated_at`, we added the following code to our model:

```
include Mongoid::Timestamps::Created
```

The following screenshot shows how the impressions will be displayed on the show page:



Writing map-reduce and aggregation to fetch and analyze data

The data is in the database now. However, we still need to read and analyze it. We will query our database in different ways and get the data based on this. We will track the total number of clicks on an article, the total number of impressions on an article, and the total number of unique impressions per day. We will use MongoDB queries and the `map-reduce` function to achieve this.

The map-reduce function is a combination of two procedures:

- ▶ **Map:** This is a procedure that filters and sorts the records
- ▶ **Reduce:** This is an operation that performs the remaining function, for example, counting

Clicks and impressions increase really quickly in huge volumes, and normal queries can be too slow; the performance could take a beating because of this. In case we need to collect our data in different ways, we can use the map-reduce function.

Engage thrusters

Let us first work with getting the data for the number of clicks and then for the number of impressions in this task:

1. In order to get the number of clicks, we will get all the clicks associated with a particular article and count them. This is an instance method:

app/models/article.rb

```
def track_clicks_per_article
  clicks = Click.where(article_id: "#{self.id}")
  click_count = clicks.count

end
```

2. For a logged-in user, we can display the click count on the article's show page; however, this will be displayed only to the logged-in user. The following code describes how we do this inside the show method:

app/controllers/articles_controller.rb

```
def show
  impressionist(@article, message: "A User has viewed your
  article")

  url = request.fullpath.to_s
  ip = request.remote_ip

  if user_signed_in? && (current_user.id != @article.user_id)

    @clicks = @article.track_clicks_per_article
    Click.record(url, ip, @article.id, current_user.id.to_s)

  elsif !user_signed_in?
```

```
    Click.record(url, ip, @article.id, "anonymous")  
  end  
end
```

3. In `show.html.erb`, `@clicks` displays the number of clicks:

```
<% if user_signed_in? %><%= @clicks %> clicks so far!<% end %>
```

4. In order to count the daily clicks, we will use the map-reduce function of MongoDB. We will first write the map function. The `this.created_at` and `this.article_id` methods will basically select these fields from the click collection. They will also initiate a count:

`app/models/click.rb`

```
def self.clicks_per_article_per_day  
  map = %Q{  
    function() {  
      emit({created_at: this.created_at, article_id: this.article_id}, {count: 1});  
    }  
  }  
end
```

5. Our reduce function will count the number of times `article_id` has occurred on a `created_at` date. This will generate an array with a daily count of clicks:

`app/models/click.rb`

```
reduce = %Q{  
  function(key, values) {  
    var count = 0;  
    values.forEach(function(v) {  
      count += v['count'];  
    });  
    return {count: count};  
  }  
}
```

6. Finally, we will run map-reduce and return the value in a variable form:

`app/models/click.rb`

```
def self.clicks_per_article_per_day  
  map = %Q{  
    function() {
```

```
        emit({created_at: this.created_at, article_id: this.article_id}, {count: 1});
    }
}

reduce = %Q{
  function(key, values) {
    var count = 0;
    values.forEach(function(v) {
      count += v['count'];
    });
    return {count: count};
  }
}
click_count = self.map_reduce(map, reduce).out(inline: true)
return click_count
end
```

7. We will fire up the console now and try to run map-reduce:

```
1.9.3-p327 :004 > @daily_clicks = Click.clicks_per_article_per_day
=> #<Mongoid::Contextual::MapReduce
  selector: {}
  class:     Click
  map:
    function() {
      emit({created_at: this.created_at, article_id: this.article_id}, {count: 1});
    }
    reduce:
    function(key, values) {
      var count = 0;
      values.forEach(function(v) {
        count += v['count'];
      });
      return {count: count};
    }
  finalize:
  out:       {:inline=>true}>
```

8. Since `daily_clicks` is an array, we will use the `each` method to loop over it and print the clicks on our command line:

```
MOPED: 127.0.0.1:27017 COMMAND      database=project6_
development command={:mapreduce=>"clicks", :map=>"\n  function()
{\n    emit({created_at: this.created_at, article_id: this.
```

```
article_id}, {count: 1}); \n    } \n  }, :reduce=>"\nfunction(key, values) {\n  var count = 0;\n  values.\n  forEach(function(v) {\n    count += v['count'];\n  });\n  return {count: count};\n} \n  }, :query=>{},\n:out=>{:inline=>true}} runtime: 112.6887ms\n{_id=>{"created_at"=>#<BSON::Undefined:0x0000000399fd88>,\n"article_id"=>BSON::ObjectId('528011687277750d4a000000')},\n"value"=>{"count"=>22.0}}\n{_id=>{"created_at"=>2013-11-13 15:18:00 UTC, "article_id"=>BSON\n::ObjectId('528011687277750d4a000000')}, "value"=>{"count"=>1.0}}\n{_id=>{"created_at"=>2013-11-13 15:18:00 UTC, "article_id"=>BSON\n::ObjectId('528011687277750d4a000000')}, "value"=>{"count"=>1.0}}\n{_id=>{"created_at"=>2013-11-13 23:14:43 UTC, "article_id"=>BSON\n::ObjectId('528011687277750d4a000000')}, "value"=>{"count"=>1.0}}\n{_id=>{"created_at"=>2013-11-13 23:56:30 UTC, "article_id"=>BSON\n::ObjectId('528011687277750d4a000000')}, "value"=>{"count"=>1.0}}\n{_id=>{"created_at"=>2013-11-13 23:56:30 UTC, "article_id"=>BSON\n::ObjectId('528011687277750d4a000000')}, "value"=>{"count"=>1.0}}
```

9. In order to track the daily impressions, we will essentially use the same functions. The only difference here is that we will define it in the `page_impression` model, as we have already included the `impressionist` models in it:

```
app/models/page_impression.rb\n\ndef self.unique_impressions_per_day\n  map = %Q{\n    function() {\n      emit(this['_id']['created_at'], {count: 1});\n    }\n  }\n  reduce = %Q{\n    function(key, values) {\n      var count = 0;\n      values.forEach(function(v) {\n        count += v['count'];\n      });\n      return {count: count};\n    }\n  }\n}
```

```
        }

        unique_impressions = self.map_reduce(map, reduce).out(inline:
true)

        return unique_impressions

    end
```

Objective complete – mini debriefing

In this task, we started by counting the number of clicks on a particular article. We created a map-reduce function to count the number of unique impressions created on a daily basis. The first part of the map-reduce function is map. It is basically a function that creates an association between a key and a value and emits the key-value pair subsequently:

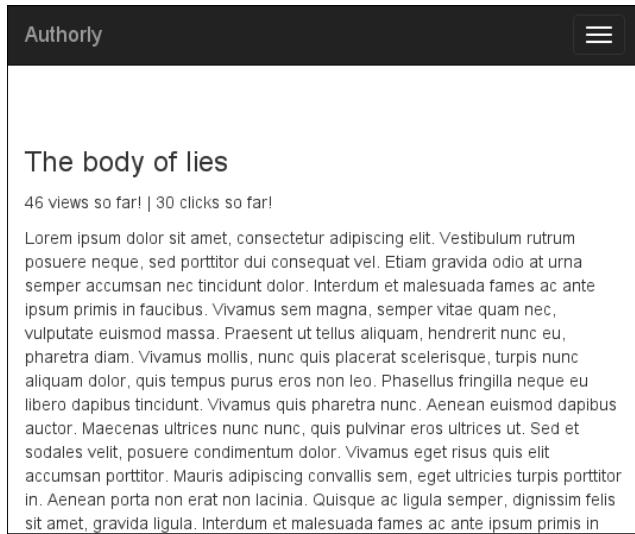
```
map = %Q{
  function() {
    emit({created_at: this.created_at, article_id: this.article_id},
{count: 1});
  }
}
```

The map function shown in the preceding example emits the key-value pairs and its value using created_at and similar other pairs. The this attribute in this.created refers to the document on which map-reduce is supposed to run; in this case, PageImpression. So, we see that the this function is exactly the same as the self function. After that, the reduce function basically reads the key and value and counts the occurrences of the key-value pairs to return the count. We then initialize the count at zero (0) and increment it as and when we hit the identical values:

```
reduce = %Q{
  function(key, values) {
    var count = 0;
    values.forEach(function(v) {
      count += v['count'];
    });
    return {count: count};
  }
}
```

The map-reduce function is a practice used very specifically for extremely large datasets. For relatively smaller datasets, it might be an overkill. Also, map-reduce generates an array of objects as a result of this. We have to loop over this array and extract the value of the click attributes from it. We also use map-reduce to find the unique impressions per day. Some of the other use cases include a data clustering, distributed data processing, and search based on specific patterns in the use cases.

The following screenshot displays the count of both clicks and impressions on the article's show page:



Creating a dashboard to display clicks and impression values

Until now, we have created various ways in the previous tasks to record, calculate, and analyze the data. As a result, we now have the data and also the count of clicks as well as impressions, and we need a dashboard to display these values. In this task, we will create a dashboard for this purpose. We have to create a dashboard controller and an admin namespace similar to the one we created in our previous project.

Engage thrusters

In the following steps, we will add an admin dashboard to the application:

1. In `dashboard_controller`, we will call all the articles:

```
app/controllers/admin/dashboard_controller.rb
class Admin::DashboardController < ApplicationController
  before_filter :authenticate_user!

  def index
    @articles = Article.all
  end
end
```

2. We will now loop over these articles and call on the methods to calculate clicks on each article:

```
app/views/admin/dashboard/index.html.erb
<h3>Clicks and Impressions Per article</h3>
<table class="table"><thead><tr><th>Article</th><th>Clicks</th></tr></thead>

<tbody><% @articles.each do |article| %><tr><td><%= link_to
article.title, article %></td><td><%= article.track_clicks_per_
article %></td></tr><% end %></tbody>
</table>
```

3. We will also count the number of impressions and display them in the table:

```
app/views/admin/dashboard/index.html.erb
<h3>Clicks and Impressions Per article</h3>
<table class="table"><thead><tr><th>Article</th><th>Clicks</th><th>Impressions</th></tr></thead>

<tbody><% @articles.each do |article| %><tr><td><%= link_to
article.title, article %></td><td><%= article.track_clicks_per_
article %></td><td><%= article.impressions_count %></td></tr><%
end %></tbody>
</table>
```

Objective complete – mini debriefing

We have created a table to display all the articles and the corresponding values of clicks and impressions on them. This is one part of the reporting structure that we're going to provide to the content creators. In the next tasks, we're going to plot our data and make better looking reports for our system.

Article	Clicks	Impressions
The body of lies	31	46
Another test article	3	3

Creating a line graph of the daily click activity

For content creators, "clicks per day" is a very important metric. They love to see the interaction and engagement happening on a day-to-day basis. We can plot the click data for the authors of the articles using the `morris.js` charts where `morris.js` is a library for plotting the data as line charts, bar charts, and donut charts. This is the reporting part of our analytics dashboard.

Engage thrusters

We will now plot the data that we have collected and analyzed in our previous tasks:

1. The `morris.js` library comes packaged as a gem. It also depends on an SVG that renders a canvas library called `raphael.js`.

```
Gemfile
gem 'morrisjs-rails'
gem 'raphael-rails'
```

2. We will add this to the `Gemfile` and run `bundle`.
3. We will then define the JavaScript in our `application.js` file. We have to ensure that these lines are placed before `require turbolinks` and `require_tree`:

```
app/assets/javascripts/application.js
//= require raphael
//= require morris
//= require turbolinks
//= require_tree .
```

4. Also, we will add the `morris.js` style sheet to our asset pipeline:
5. In order to feed data to the JavaScript charts, we will have to prepare our data in the JSON format. To do this, first call the `clicks_per_article_per_day` method. As you can see, we have created a new method called `clicks` for this:

```
app/controllers/admin/dashboard_controller.rb
def clicks
  @daily_clicks = Click.clicks_per_article_per_day
end
```

6. We need the count of clicks and the date in order to plot this graph. Hence, we will get the results of the `clicks_per_article_per_day` method and generate a json hash for `morris.js` to read. For this, we will first create a model class method that loops over the data to generate a hash:

```
app/models/click.rb
def self.get_click_data
    daily_clicks = self.clicks_per_article_per_day
    click_data = []
    daily_clicks.each do |d|
        id = d["_id"]
        daily_clicks = d["value"]
        date = d["_id"]["created_at"]
        clicks = daily_clicks["count"]
        click_data << { :date => date.to_i, :clicks => clicks.to_i}
    end
    return click_data
end

app/controllers/admin/dashboard_controller.rb
def clicks
    @click_data= Click.get_click_data
    respond_to do |format|
        format.json { render json: @click_data }
    end
end
```

7. Despite the availability of data in JSON, we need a way to access it. So, we will write a route to access the data using the `this` method.

```
config/routes.rb
namespace :admin do
    get '', to: 'dashboard#index', as: '/'
    get "dashboard(clicks"
end
```

8. In our `app/views/admin/dashboard/index.html.erb` file, we will initiate the script for clicks. The `Morris.Line` function is a function to create a line graph. We will keep the date as the key for the x axis and clicks as the key for y axis:

```
app/views/admin/dashboard/index.html.erb
<script>
var url = "/admin/dashboard(clicks.json"
var click_json= $.getJSON(url, null, function(data) {
```

```

var get_click_data = click_json.responseText;

new Morris.Line({
  element: 'click_chart',
  data: $.parseJSON((get_click_data)),
  xkey: 'date',
  ykeys: ['clicks'],
  labels: ['Clicks']
});
done();
});


```

- Lastly, we will render this in a `div` tag. Make sure `<div id>` and the element name in the `Morris.Line` definition are the same:

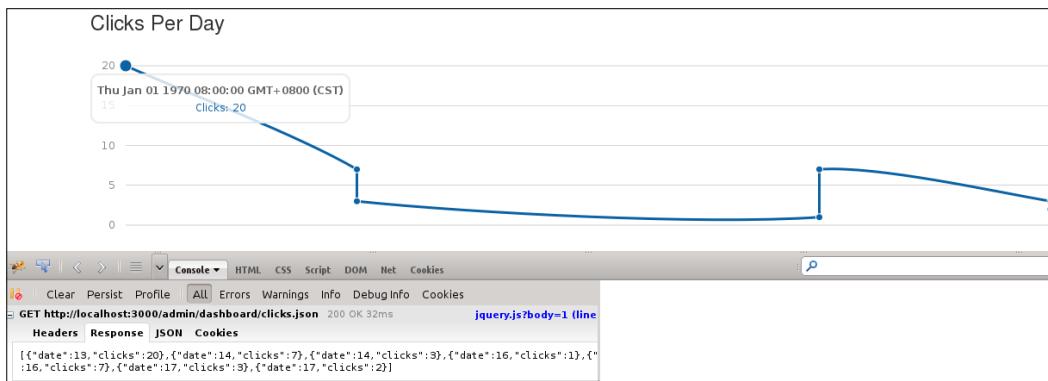
```

app/views/admin/dashboard/index.html.erb
<h3>Clicks Per Day</h3>
<div id="click_chart" style="height: 250px;"></div>

```

Objective complete – mini debriefing

The previous task included the creation of JSON from the data we already have, and `morris.js` accepts this data in a particular format. We had to extract the data from our map-reduce function and format it according to the format accepted by `morris.js`. Please see the following screenshot for the acceptable format:



You will notice that `date` is in the integer format because or to get the date, we did the following in our `map` function:

```

map = %Q{
  function() {

```

```
        emit({created_at: this.created_at.getDate()}, {count: 1});  
    }  
}
```

The `getDate()` function will return the date in the float format. In order to render it on the frontend, we will convert the float datatype to the integer datatype:

```
@click_data <- { :date => date.to_i, :clicks => clicks.to_i }
```

This method generates json, which can be directly read by visiting the `/admin/dashboard/clicks.json` URL. To display the clicks, we made a call on the `clicks.json` data by directly calling this URL:

```
var url = "/admin/dashboard/clicks.json"
```

To extract the data from json, we will use the `function(data)` jQuery method and store the data text in `get_click_data`:

```
var click_json= $.getJSON(url, null, function(data) {  
    var get_click_data = click_json.responseText;  
    click_json.responseText;
```

Finally, we passed the data to the `Morris.Line` method to generate the line graph. The `morris.js` line graph accepts `xkey` and `ykeys` as axes and `labels` to represent data at each data point. You can set colors, customize the text, set the line width, and set data formats and units for each datapoint:

```
new Morris.Line({  
    element: 'click_chart',  
    data: $.parseJSON((get_click_data)),  
    xkey: 'date',  
    ykeys: ['clicks'],  
    labels: ['Clicks']  
});  
});  
});  
Insert
```

Creating a bar graph of the daily visit activity

In the previous task, we already learned how to display the daily click data on a line graph. In this task, we will use bar charts to display the daily visit activity of the impression data. We will also create json from the impression data we have and feed it to the `morris.js` method to generate our graph.

Engage thrusters

We will now use the following steps to create a bar chart of the impression data:

1. In `dashboard_controller`, we will create a method called `impressions` to construct the impressions JSON:

```
app/controllers/admin/dashboard_controller.rb
def impressions
  @daily_impressions = Article.impressions_per_article_per_day
end
```

2. In the `article` model, we will edit our `map` method and change the format of `created_at` to `getDate()`:

```
app/models/article.rb
def self.impressions_per_article_per_day
  map = %Q{
    function() {
      emit({created_at: this.created_at.getDate()}, {count: 1});
    }
  }
end
```

3. In the `impressions` method, we will construct JSON and render it:

```
app/controllers/admin/dashboard_controller.rb
def impressions
  daily_impressions = Article.impressions_per_article_per_day
  @impressions_data = []

  daily_impressions.each do |d|
    id = d["_id"]
    daily_impressions = d["value"]
    date = d["id"]["created_at"]
    impressions = daily_impressions["count"]
    @impressions_data << { :date => date.to_i, :impressions => impressions.to_i }
  end
  respond_to do |format|
    format.json { render json: @impressions_data }
  end
end
```

4. We will tie this to a route in order to generate the URL:

```
config/routes.rb
namespace :admin do
  get '', to: 'dashboard#index', as: '/'
```

```
get "dashboard-clicks"
get "dashboard/impressions"
end
```

5. The function to generate a bar graph is quite similar to the one for a line graph. The axis key definitions are also the same:

```
app/views/admin/dashboard/index.html.erb
<script>
var url = "/admin/dashboard/impressions.json"
var json=json= $.getJSON(url, null, function(data) {
var get_impression_data = json.responseText;json.responseText;

new Morris.Bar({
  element: 'impressions_chart',
  data: $.parseJSON((get_impression_data)),
  xkey: 'date',
  ykeys: ['impressions'],
  labels: ['Impressions']
});
});
</script>
```

6. With the JavaScript method ready to create a bar chart, we just need to render our graph:

```
<h3>Impressions Per Day</h3>
<div id="impressions_chart" style="height: 250px;"></div>
```

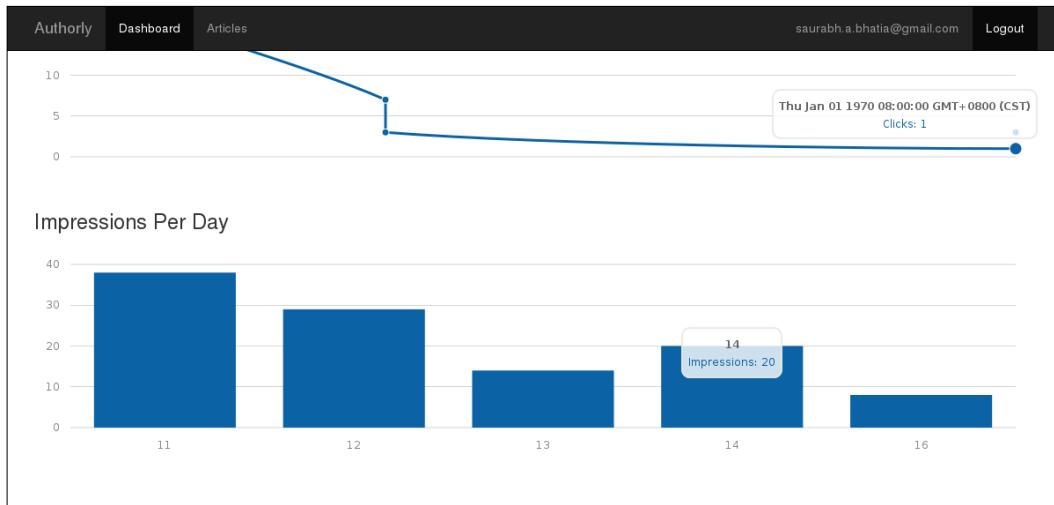
Objective complete – mini debriefing

In this task, we used a bar chart to represent the impression data that we collected in our previous tasks. We used the same method as clicks to generate JSON. We used `Morris.Bar` to generate the bar graph. We used `xkey` and `ykeys` to represent the `x` and `y` axes. We used `labels` to represent data against each bar. Some of the other options provided in the `morris.js` bar graph are as follows:

- ▶ You can enable or disable grid lines by setting the `grid` option to `true` or `false`
- ▶ You can enable or disable the display of axes by setting the `axes` option to `true` or `false`
- ▶ To manipulate the text properties of the grid you have `gridTextColor`, `gridTextSize`, and `gridTextWeight`
- ▶ You have a `stacked` option—a Boolean value—to allow bars to be vertically stacked
- ▶ You have a `BarColors` option, which is an array to set the colors of the bars

- ▶ You have a HideHower option to show or hide the data on Hower
- ▶ You also have a HowerCallback option that allows additional functions to generate custom hovers

The following screenshot shows a bar graph:



Creating a demographic-based donut chart

We have already plotted our click and impression data as both a line and bar graph. However, we also have to track the demographics of our user visits. One of the parameters for demographics is the location of the visitor. As a part of our requests, we can easily track the country and city of the user based on the user's IP address. We will add these methods for our tracking mechanisms and generate a donut chart to visualize our visitor's locations.

Prepare for lift off

In order to proceed with this section, we will be using a Geocoder to track the location of the visitor. A Geocoder is a very comprehensive library to not only locate the user and get the coordinates, but also run the Geospatial queries; for example, to find nearby users. For this, we will add the geocoder gem to Gemfile and run bundle install:

```
Gemfile
gem 'geocoder'
```

Engage thrusters

The following steps include the methods to generate and represent the demographic data of our visitors:

1. In order to get the demographics, we need to get the country data. In order to record the country data, we will add two fields to our click collection:

```
app/models/click.rb  
field :country, type: String
```

```
field :city, type: String
```

2. We will add `request.location.country` and `request.location.city` to our show method inside `articles_controller.rb`. We will also save these as part of our click objects:

```
app/controllers/articles_controller.rb  
def show  
    @country = request.location.country  
    @city = request.location.city  
  
    click.country = @country  
    click.city = @city  
end
```

3. So, our final show method will look like the following code:

```
app/controllers/articles_controller.rb  
def show  
    @clicks = @article.track_clicks_per_article  
    impressionist(@article, message: "A User has viewed your  
article")  
    @url = request.fullpath.to_s  
    @ip = request.remote_ip  
    @country = request.location.country  
    @city = request.location.city  
    url = request.fullpath.to_s  
    ip = request.remote_ip  
    country = request.location.country  
    city = request.location.city  
    if user_signed_in? && (current_user.id != @article.user_id)  
        Click.record(url, ip, country, city, @article.id, current_  
user.id.to_s)  
    elsif !user_signed_in?
```

```
    Click.record(url, ip, country, city, @article.id,
"anonymous")
end
end
```

4. Also, we will modify our `record_data` concern to save these values to the database:

```
app/models/concerns/record_data.rb

module RecordData

  extend ActiveSupport::Concern

  included do

    def self.record(url, ip, country, city, article_id, user_id)
      self.create!(url: url, ip: ip, country: country, article_id:
article_id, user_id: user_id)
    end
  end
end
```

5. Once we have the mechanism set up to record the data, we will write a map-reduce function to count the number of visits from a particular country:

```
app/models/click.rb

def self.clicks_per_country
  map = %Q{
    function() {
      emit({country: this.country}, {count: 1});
    }
  }

  reduce = %Q{
    function(key, values) {
      var count = 0;
      values.forEach(function(v) {
        count += v['count'];
      });
      return {count: count};
    }
  }
  unique_clicks = self.map_reduce(map, reduce).out(inline: true)
  return unique_clicks
end
```

6. In `dashboard_controller`, we will add a `demographics` method to generate JSON for our recorded data:

```
def demographics
  demographics = Click.clicks_per_country
  @impressions_data = []

  demographics.each do |d|
    id = d["_id"]
    demographics = d["value"]
    country = id["country"]
    visits = demographics["count"]
    @impressions_data << { :country => country, :visits =>
  visits.to_i}

  end
  respond_to do |format| format.json { render json: @
impressions_data }
  end
end
```

7. We will add a route for this to generate the URL:

```
config/routes.rb
namespace :admin do
  get '', to: 'dashboard#index', as: '/'
  get "dashboard-clicks"
  get "dashboard/impressions"
  get "dashboard/demographics"
end
```

8. We will initialize a donut chart to display this data:

```
app/views/admin/dashboard/index.html.erb
var url = "/admin/dashboard/demographics.json"
var demographic_json=demographic_json= $.getJSON(url, null,
function(data) {
  var get_demographic_data = demographic_json.responseText;

  Morris.Donut ({
    element: 'demographic',
    data: get_demographic_data
  });

```

9. Lastly, display the chart in div:

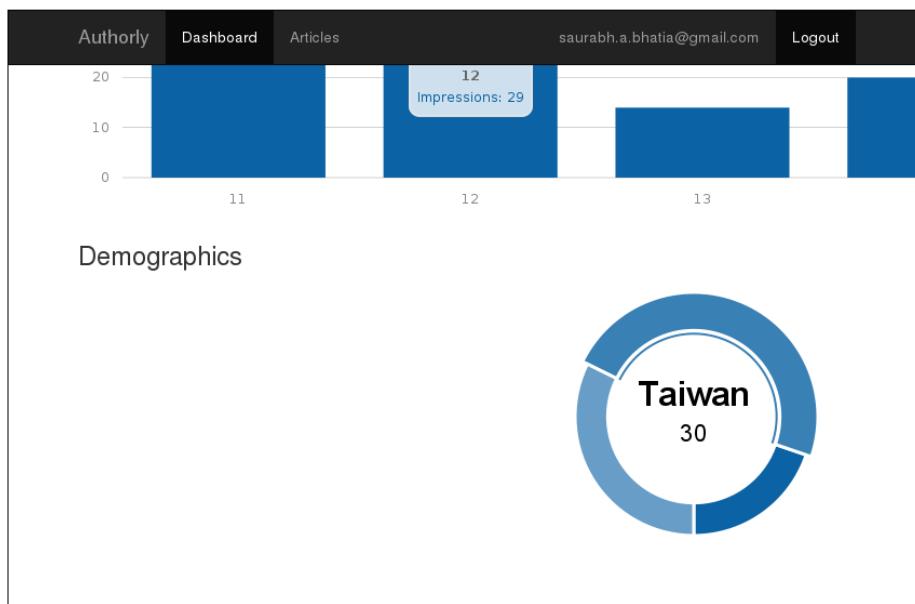
```
app/views/admin/dashboard/index.html.erb
<h3>Demographics</h3>
<div id="demographic" style="height: 250px;"></div>
```

Objective complete – mini debriefing

We first used the `request.location.country` and `request.location.city` methods to look for the country and city based on the IP address of the visitor. These methods were available as soon as we bundled the geocoder gem in our applications. We wrote a map-reduce function to count the number of visits from a particular country. The `map` function aggregated all the impressions based on the country and the `reduce` function in this case counted the size of each aggregation.

A donut chart is very similar to a pie chart. In our case, it represents the break up of visits from a particular country. We created a method called `demographic` in our dashboard controller. We generated a json hash that included demographics that were consumed by the `morris.js` donut chart method. Donut charts do not have axes. The data here is represented as a label and a value. It also accepts the colors and formatter parameters. Colors contain the HTML color code for the donut segment.

The following is how a donut chart looks when generated using `morris.js`:



Mission accomplished

We have created a fully functional analytics dashboard in this project. As mentioned earlier, the analytics dashboard has three main parts:

- ▶ **Recording:** We created a mechanism to track clicks, visits or impressions, and demographics of the user
- ▶ **Analyzing:** We wrote various queries and `map-reduce` methods to count visits, clicks, unique visits, and visits from each country
- ▶ **Reporting:** We created tables and charts of different types in order to represent and visualize the data we recorded and analyzed

Hotshot challenges

In an analytics dashboard, the possibilities are endless as to how you can imagine the data. We can improve our dashboard with some exercises:

- ▶ Write `map-reduce` to make a leaderboard for articles and display the top 10 articles
- ▶ Create localized slugs for our articles
- ▶ Use `ensureindex` to create an index and improve the performance of the impressionist query
- ▶ Display the article names on the line and bar charts
- ▶ Create an area chart to compare the activities of the top three articles by a particular user

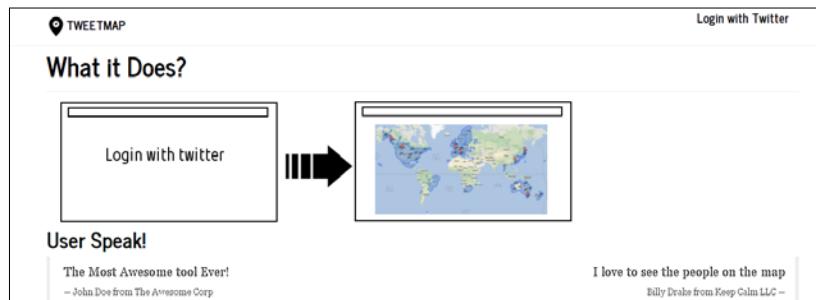
Project 7

Creating an API Mashup – Twitter and Google Maps

Social media is an important tool these days, and with the developer APIs available for most services such as Facebook, Twitter, and Google, the possibilities are endless. There are so many applications of these APIs, especially when you do not want a user to create a new login and when you want to give your application a social twist by sharing the data from multiple social networks inside your application.

Mission briefing

In this project, we will create an application that utilizes Twitter and Google Maps API. We will use Twitter OAuth2 to authenticate the user using Twitter, and we will use Google Maps API v3 to display the friends of the user on a Google map. We will visualize the location of the user's Twitter friends using this application. As shown in the following screenshot, we will see our friends with their corresponding locations on the map:



Why is it awesome?

APIs are an important part of many web applications nowadays. It not only builds a loyal developer community, thereby backing the web application, but also improves the user engagement with the application. Facebook, Twitter, and Google APIs are the most commonly used because of their extremely high user base, clean API methods, and a huge developer community to back them up. These APIs are also easy to include in the application through community-contributed interfaces. We will look at some of them while building this project.

At the end of this project, we will be able to mashup Twitter and Google map APIs and make a fun little application.

Your Hotshot objectives

While building this application, we will have to go through the following tasks:

- ▶ Creating an application login with Twitter
- ▶ Calling all Twitter friends
- ▶ Getting latitude and longitude details of the user's location
- ▶ Passing Twitter data to the Google Maps API using Rails
- ▶ Displaying friends on the map using the Google API
- ▶ Creating points of interest—filter users based on their location

Mission checklist

We need the following installed on the system, and we also need to sign up for the API keys before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MongoDB
- ▶ Bootstrap 3.0
- ▶ Sass
- ▶ Devise
- ▶ Twitter API keys
- ▶ Google Maps API keys
- ▶ Git
- ▶ A tool for mock-ups
- ▶ jQuery

Creating an application login with Twitter

In the first task, we will create a login using Twitter and allow the users to authenticate using this. We will use the `omniauth` gem and add some custom methods in order to handle the session. OmniAuth is a solution for authentication that uses rack via multiple third-party OAuth providers such as Google, Twitter, Facebook, and GitHub. The `omniauth` gem (<https://github.com/intridea/omniauth>) provides the rack-based methods of authentication and sessions. Individual access methods for each provider is called a strategy. Each strategy is extracted into different gems. So, if we want to implement Twitter and Facebook, we need three gems: `omniauth`, `omniauth-twitter`, and `omniauth-facebook`.

Prepare for lift off

Before we start the work on this project, we will have to sign up for the API keys on Twitter and Google. Log in to Twitter as a developer and create an application by navigating to <https://dev.twitter.com/apps/new>. The page will look like the following screenshot:

The screenshot shows a web browser window with the URL <https://dev.twitter.com/apps/new> in the address bar. The page title is "Create an application". The main section is titled "Application Details" and contains three fields: "Name:" with a placeholder "Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.", "Description:" with a placeholder "Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.", and "Website:" with a placeholder "Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in user-facing authorization screens for example, for user confirmation and will be shown in some contexts as the application owner".

Once we submit the form after filling in the details, it will generate an application token and an application secret for us. As a part of our application details, we need to fill a field called **Callback URL**. Callback is defined as the URL where Twitter sends back the session details after you log in. By design, Twitter API does not support localhost, so in order to work with the application locally, we will define the **Callback URL** as `http://lvh.me:3000`. We have seen the various ways in which this dummy domain is used in *Project 4, Creating a Restaurant Menu Builder*.

The screenshot shows the Twitter Developers website with the 'Developers' tab selected. The 'OAuth settings' section is displayed, containing a table of application details:

Setting	Value
Access level	Read-only About the application permission model
Consumer key	Rd5s5s82FAiUD1KufnrrQ
Consumer secret	6q8LouMcq8qE1aZa5Mn5nONDwpzchrmXOIlqEYl9CU
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token
Callback URL	http://lvh.me:3000/
Sign in with Twitter	No

Engage thrusters

We will take the first steps in this task to set up the base of the application:

1. We will install `omniauth` and `omniauth-twitter`, the Twitter strategy gem from the master branch, by adding it to the `Gemfile` and run `bundle install`, as shown in the following code:

```
Gemfile
gem 'omniauth'
gem 'omniauth-twitter', :github => 'arunagw/omniauth-twitter'
tweetmap$ bundle install
```

2. We will create a file called `secrets.yml` inside the `config` folder. This file should contain `secret_key_base` and all the secret keys to be used in the app. We will explore this feature in detail in our debriefing section. Make sure you generate a different set of keys for development and production:

```
config/secrets.yml
development:
  secret_key_base: APPLICATION_SECRET TOKEN
  twitter_consumer_key: CONSUMER_KEY
  twitter_consumer_secret: CONSUMER_SECRET

  test:
    secret_key_base: APPLICATION_SECRET
    _TOKEN
    twitter_consumer_key: CONSUMER_KEY
    twitter_consumer_secret: CONSUMER_SECRET

  production:
    secret_key_base: APPLICATION_SECRET
    _TOKEN
    twitter_consumer_key: CONSUMER_KEY
    twitter_consumer_secret: CONSUMER_SECRET

config/initializers/omniauth.rb
Rails.application.config.middleware.use OmniAuth::Builder do
  provider :twitter, Rails.application.secrets.twitter_consumer_
key, Rails.application.secrets.twitter_consumer_secret
end
```

3. We will generate a model for the user. This model will hold the values for the provider (Twitter), such as the name of the user, the screen name, or the Twitter handle, `oauth_token`, `expires_at` (expiration time of `oauth_token`), and location of the user:

```
tweetmap$rails g model user provider:string uid:string name:string
oauth_token:string oauth_secret:string oauth_expires_at:datetime
avatar:string address:string
```

4. Our migration looks like the following code:

```
20131123144240_create_users.rb
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :provider
      t.string :uid
      t.string :name
      t.string :oauth_token
```

```
    t.string :oauth_secret
    t.string :avatar
    t.string :address
    t.datetime :oauth_expires_at
    t.timestamps
  end
end
end
```

5. In our `user` model, we will access certain values from the Twitter API's response hash and store it in the `user` table we just created:

```
app/models/user.rb
class User < ActiveRecord::Base
  def self.create_with_omniauth(auth)
    create! do |user|
      user.provider = auth["provider"]
      user.uid = auth["uid"]
      user.name = auth["info"]["name"] || ""
      user.address = auth["info"]["location"] || ""
      user.avatar = auth["info"]["image"] || ""
      user.oauth_token = auth["credentials"]["token"] || ""
      user.oauth_secret = auth["credentials"]["secret"] || ""
    end
  end
end
```

6. After adding it to the `user` model, we need a mechanism to get these values. This is possible only when we are able to start a session with Twitter.
7. To set up and handle a Twitter session, we will need a controller for sessions called `session_controller.rb`. We will add methods to create and destroy the session, that is, the `signup`, `login`, and `sign out` options:

```
tweetmap$ rails g controller sessions
app/controllers/session_controller.rb
class SessionsController < ApplicationController
  def create
    auth = request.env["omniauth.auth"]
    user = User.find_by_provider_and_uid(auth["provider"], auth["uid"]) || User.create_with_omniauth(auth)
    session[:user_id] = user.id
    redirect_to root_url, :notice => "Logged In Successfully"
  end
  def destroy
    session[:user_id] = nil
  end
end
```

```
    redirect_to root_url, :notice =>"Logged Out Successfully"
  end
end
```

8. For the controller to work, we need to add the routes in our `routes.rb` file:

```
config/routes.rb
match "/auth/:provider/callback" =>"sessions#create", via: [:get,
:post]
  match 'signout', to: 'sessions#destroy', as: 'signout', via:
[:get, :p]
```

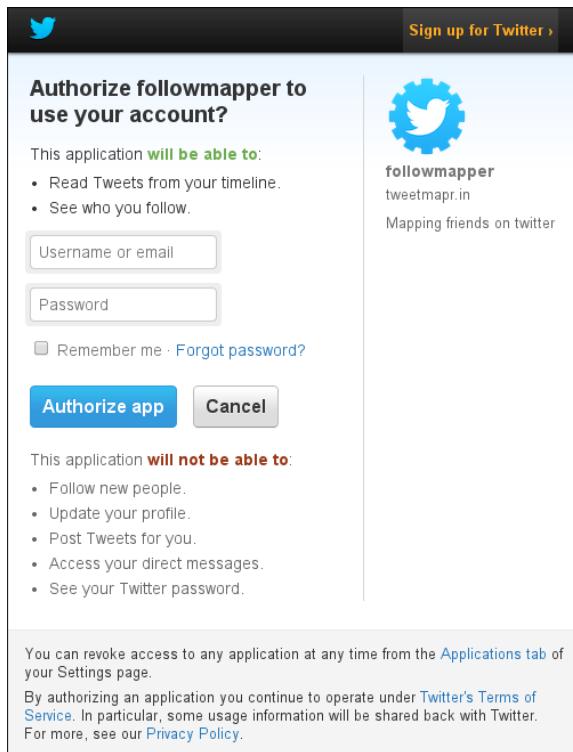
9. Now that we have created a session, we will have to add a method to access the user object while in the session. We will do this by creating an object called `current_user` in our `application_controller.rb` file:

```
app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
  helper_method :current_user
  private
  def current_user
    @current_user ||= User.find(session[:user_id]) if
session[:user_id]
  end
end
```

10. Also, we need to create a link to log in using Twitter. In our `views/layouts/application.html.erb` file, we will add a Sign In with Twitter link:

```
app/views/layouts/application.html.erb
<div class="navbar-collapse collapse" id="navbar-main">
<ul class="nav navbar-nav navbar-right">
<% if current_user %>
<li>Welcome, <%= current_user.name %><%= image_tag "#{current_
user.avatar}" %><%= link_to "Sign Out", signout_path %></li>
<% else %>
<li><%= link_to "Sign in with Twitter", "/auth/twitter" %></li>
<% end %>
</ul>
</div>
```

11. We will now click on **Sign in with Twitter** and see where it takes us. Once we do this, we are presented with the Twitter login screen as shown in the following screenshot:



Objective complete – mini debriefing

This task dealt with the addition of OmniAuth to the application. OmniAuth supports all the major services such as Facebook, Twitter, and Google. In the current version of OmniAuth, that is 1.2.1, we need to add the `omniauth` gem and also the gem that supports the respective provider strategy. In our case, the provider strategy uses Twitter. The same `user` table can be used to implement Facebook and Google strategies too.

In Rails 4.1, there is a new way to handle all the API keys and secrets in a much more secure way. When you generate a new project in Rails 4.1, Rails generates a `secrets.yml` file for us. This is a replacement to `secret_token.rb` that was earlier generated inside `config/initializers`. In Rails 3.2, the parameter was called `secret_token` too. In Rails 4, this has been renamed to `secret_key_base` and moved to a completely different file. We added lines for Twitter credentials in the `secrets.yml` file:

```
twitter_consumer_key  
twitter_consumer_secret
```

In order to access the value of the preceding field in the controller, we can directly call `Rails.application.secrets` followed by the name of the field:

```
Rails.application.secrets.twitter_consumer_key
```

We then created a model for user and table to store all the callback values. Twitter or any API that uses OAuth for authentication returns `oauth_token` and `oauth_expires_at`. The token is a unique string that expires after a particular time interval of idleness. This is to terminate the session when not in use and keep the token from being stolen. To save the user to the database and create a session, we ran the `create_with_omniauth` method with the `auth` hash as an argument:

```
def self.create_with_omniauth(auth)
```

We created a controller to handle sessions against provider's Twitter user ID. This method works similar to the `find_or_create_by` method in Rails. It looks for the presence of the user ID and provider. If found, it creates a session; otherwise, it asks for permission to allow or reject the application from the user ID.

We then set the `current_user` object and persisted it in the session. We finally added a method to handle the user object during the course of the session. In the following screenshot, we can see the user logged in with the Twitter credentials:



Calling all Twitter friends

In order to get the details of a user from Twitter, we will use the interface to the Twitter API, the `twitter` gem. In this task, we will pull some details of the user such as the Twitter username, the Twitter handle, the location of the user, and the user's avatar. We will store this information as a part of our `user` table. Friends are the users that are either followed by the user or follow the user.

Engage thrusters

We will now go ahead and access the Twitter data using the Twitter API:

1. We will first add some more columns to our `user` table with the following code:

```
tweetmap$ rails g add_details_to_users address:string  
avatar:string
```

2. The migration file that is generated looks like the following code:

```
class AddDetailsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :address, :string
    add_column :users, :avatar, :string
  end
end
```

3. We will save the link to `avatar` of the user and the user's location.
4. We will now add the `twitter` gem to the `Gemfile` and run `bundle install`, as shown in the following code:

```
Gemfile
gem 'twitter', :github => 'sferik/twitter'
```

5. We will now generate a model to save the friends' data:

```
Tweetmap$ rails g model friend name:string screen_name:string
location:string latitude:float longitude:float user_id:integer
```

6. We will also edit the migration to add decimal precision in our latitude and longitude fields:

```
class CreateFriends < ActiveRecord::Migration
  def change
    create_table :friends do |t|
      t.string :name
      t.string :screen_name
      t.string :location
      t.integer :user_id
      t.float :lat, {:precision=>10, :scale=>6}
      t.float :lng, {:precision=>10, :scale=>6}
      t.timestamps
    end
  end
end
```

7. We will first create a home controller with an index action:

```
tweetmap $ rails g controller home index
```

8. In our home controller, we will create a client for our Twitter API. This will require the consumer key and consumer secret. Twitter supplies the OAuth token and OAuth secret as a part of the session parameters. We also need to initiate this in order to get the data related to the user's friends:

```
app/controllers/home_controller.rb
def fetch_friend_data
```

```
client = Twitter::REST::Client.new do |config|
  config.consumer_key         = "Rd5s5s82FAiUD1KufnrrnQ"
  config.consumer_secret      =
"6q8LouMcq8qE1aZa5Mn5nONDwpzchrmXOIlqEYl9CU"
  config.access_token          = "#{current_user.oauth_token}"
  config.access_token_secret   = "#{current_user.oauth_secret}"
end
end
```

9. We will make a call on the Twitter API to fetch the last 20 friends of the user who is logged in:

```
app/controllers/home_controller.rb
@friends = client.friends.take(20)
```

10. We will create a class method in which the user ID, the array of the friend's location coordinates, and the friend object will be passed as arguments. This method will save the friends' data to the friends table in the database:

```
app/models/friend.rb
def self.get_friend_data(friend,location_value,user_id)
  self.where(
    name: friend.name,
    screen_name: friend.screen_name,
    location: friend.location,
    user_id: user_id).first_or_create
end
```

11. We will loop through the friends' data, geocode their location, and get the coordinates. We will save these values to the database:

```
app/controllers/home_controller.rb
@friends.each do |f|
  location = f.location
  Friend.get_friend_data(f,current_user.id)
end
```

12. As you can see, we are saving the values to the database using a method called `get_friend_data` and passing some arguments to this. We need to define that method in our model:

```
app/controllers/home_controller.rb
Friend.get_friend_data(f,current_user.id)
```

13. We will create a route and a link to run this from the home page, as shown in the following code:

```
config/routes.rb
get "home/fetch_friend_data"
```

```
app/views/home/index.html.erb
<div class="row">
<div class="col-lg-6">
<h2 id="type-blockquotes"><%= link_to "Fetch My Friends", home_fetch_friend_data_path, :class=>"btn btn-primary" %></h2>
</div>
</div>
```

14. We will first log in and then click on the **Fetch My Friends** link to fetch our friends' data.
15. In order to check whether the data is being saved correctly or not, we will query our friends table:

```
1.9.3-p327 :001 > Friend.first

Friend Load (0.5ms)  SELECT `friends`.* FROM `friends` ORDER BY
`friends`.`id` ASC LIMIT 1

=> #<Friend id: 1, name: "John Doe", screen_name: "johndoe",
location: "Christchurch, New Zealand", user_id:user_id: 1,
created_at: "2013-12-07 09:12:01", updated_at: "2013-12-07
09:12:01">
```

Objective complete – mini debriefing

In the context of Twitter, friends are all the people a user follows. In the previous task, we made a call on the Twitter API to fetch the data related to a user and saved it in the database. We added the `twitter` gem to the application and initiated a client based on the Twitter credentials we signed up for, in the beginning of the project. With these in place, we will call the friends from the Twitter API. Twitter, as a part of its API, allows very limited number of calls per hour (350) and a maximum of 180 in 15 minutes. Hence, we need to be careful about how we make a call on the data. One way to call all the friends on Twitter is to call all friends as shown in the following code:

```
client.friends.all
```

The drawback of the preceding method is that we might end up exhausting our limit, quite possibly in one go, because it makes the number of requests equal to the number of friends on Twitter. An alternative way is to call a limited number of friends, as shown in the following code:

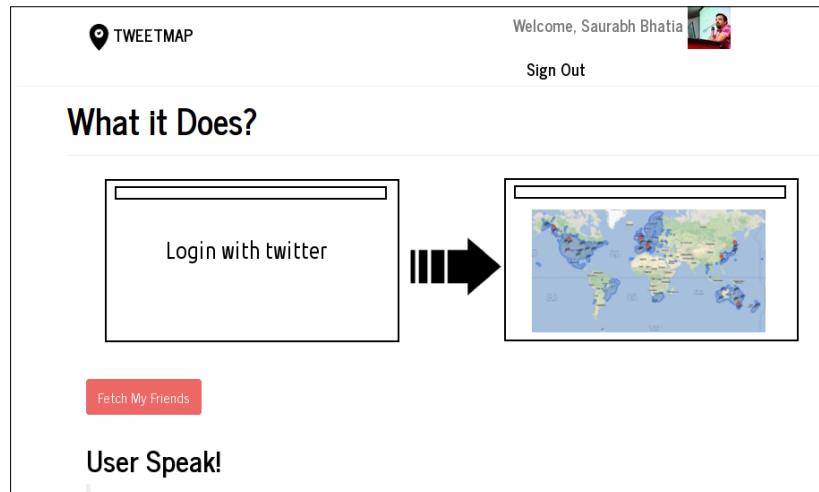
```
client.friends.take(20)
```

The preceding method issues only a single request to the Twitter API and fetches data of 20 friends in one go. This is a better way to do the same thing.

Once the friends are fetched, they need to be saved to the `friends` table. As we are making an API call, the API does not tell us that the records we are fetching are the same as previous API calls. To save the friends to the database, we will use the `find_or_create_by` method in Rails. The Rails 4 syntax is significantly different from its earlier versions. It is a combination of two ActiveRecord methods called chaining of queries as shown in the following code:

```
self.create_with(  
  name: friend.name,  
  location: friend.location,  
  latitude: location_value.first,  
  longitude: location_value.second).find_or_create_by(  
  user_id: user_id, screen_name: friend.screen_name)  
end
```

The `find_or_create_by` method looks for the user ID and screen name of the user to check whether it exists or not. If there are extra parameters that need to be checked, we can use `create_with`, which runs a `create` query in case the record is not found with the passed attributes. We finally created a route and a link to fetch the friends. The following screenshot shows the link as it would appear on the home page:



Getting latitude and longitude details of the user's location

To map the friends of the user to the map, the most important information required is the latitude and longitude. As we saw previously, Twitter provides the location of the user, and we will geocode it to find the coordinates. We will use a Ruby gem called `geocoder` in order to get this.

Engage thrusters

We will now find and save the location coordinates of our user's friends:

1. We will first add the `geocoder` gem to our `Gemfile` and run `bundle install`:
`gem 'geocoder'`
2. In the home controller, when we're saving friends, we will find the coordinates of the location using the `Geocoder.coordinates` method:

```
app/controllers/home_controller.rb
@friends.each do |f|
  location = f.location
  location_value = Geocoder.coordinates("#{location}")
  if location_value.present?
    Friend.get_friend_data(f, location_value, current_user.id)
  end
end
```

We also added the `location_value` argument that passes the coordinates to the model.

3. We will now modify the model to add the argument and save the location values with the other values:

```
app/models/friend.rb
class Friend < ActiveRecord::Base
  geocoded_by :location
  def self.get_friend_data(friend, location_value, user_id)
    self.create_with(
      name: friend.name,
      location: friend.location,
      latitude: location_value.first,
      longitude: location_value.second).find_or_create_by(
        user_id: user_id, screen_name: friend.screen_name)
  end
end
```

4. Now, we are able to save our friend's location coordinates:

```
1.9.3-p327 :001 > Friend.first

Friend Load (0.5ms)  SELECT `friends`.* FROM `friends` ORDER BY
`friends`.`id` ASC LIMIT 1

=> #<Friend id: 1, name: "John Doe", screen_name: "johndoe",
location: "Christchurch, New Zealand", user_id: 1, created_
at: "2013-12-07 09:12:01", updated_at: "2013-12-07 09:12:01",
latitude: -43.5321, longitude: 172.636>
```

5. We will also use the Geocoder.coordinates method to find the location of the user. First, we will add the migration to save our OAuth credentials:

```
$rails g migration add_omniauth_and_location_to_users
class AddCoordinatesToUsers < ActiveRecord::Migration
  def change
    add_column :users, :latitude, :string
    add_column :users, :longitude, :string
  end
end
app/model/user.rb
class User < ActiveRecord::Base
  def self.create_with_omniauth(auth)
    location = auth["info"]["location"] || ""
    user_location = Geocoder.coordinates("#{location}")
    create! do |user|
      user.provider = auth["provider"]
      user.uid = auth["uid"]
      user.name = auth["info"]["name"] || ""
      user.address = auth["info"]["location"] || ""
      user.avatar = auth["info"]["image"] || ""
      user.oauth_token = auth["credentials"]["token"] || ""
      user.oauth_secret = auth["credentials"]["secret"] || ""
      user.latitude = user_location.first
      user.longitude = user_location.second
    end
  end
end
```

Note that in order to do this step, you need to add the latitude and longitude columns to the database. The users saved will be as follows:

```
1.9.3-p327 :002 > User.first

User Load (0.6ms)  SELECT `users`.* FROM `users` ORDER BY
`users`.`id` ASC LIMIT 1
```

```
=> #<User id: 1, provider: "twitter", uid:  
"415386785", name: "Saurabh Bhatia", OAuth_token:  
"415386785-URRXAJQSyyJ1FkJQt2eSyg4hpXajoAj6PxVpzUPI", OAuth_  
expires_at: nil, created_at: "2013-12-07 09:11:49", updated_at:  
"2013-12-07 09:11:49", address: "Zhubei City, Taiwan", avatar:  
"http://pbs.twimg.com/profile_images/3408461966/ec48...", OAuth_  
secret: "oIx2ddTLd19vVj8i5xNYcxX6gtqlXu6WY14RFSXywDZJD", latitude:  
"121.0119444", longitude: "24.8333333">
```

Objective complete – mini debriefing

We used a Ruby-based geocoder API gem called `geocoder`. After we added and bundled it, we used the `Geocoder.coordinates` method to fetch the coordinates of the user's location and friends' location. In order to save it, we added the latitude and longitude columns in our `friends` table.

Another method that we can use to fetch and save the location coordinates of a friend is shown in the following code:

```
geocoded_by :location
```

The preceding method will run on an `after_save` callback to fetch the coordinates of a location. The method also fires an update query to save the latitude and longitude values in the database.

We will use the location coordinates in the upcoming tasks for various uses, such as creating markers on the Google map, and the geocoder API to create points of interest in our app.

Passing Twitter data to the Google Maps API using Rails

Now, we already have the Twitter data of the user, the data of user's friends, and also their location coordinates. From here on, we need to prepare the data to be displayed on the Google map. We need to display multiple markers on the map and associate our data with the markers.

Engage thrusters

In this task, we will prepare the data for the map:

1. We will start by creating a controller for the map. This controller will be responsible for passing the data required for the map to the Google Maps JavaScript API:

```
tweetmap$ rails g controller map_display index
```

2. The markers include four types of data: the screen name, the name, the latitude, and longitude. Before that, we will initiate a blank array:

```
app/controllers/map_display_controller.rb
class MapDisplayController < ApplicationController
  def index
    @markers = []
  end
end
```

3. We will first find all the friends of the current user:

```
app/controllers/map_display_controller.rb
class MapDisplayController < ApplicationController
  def index
    @markers = []
    @friends = current_user.friends
  end
end
```

4. We will loop over the friends data and call the screen_name, name, latitude, and longitude from it. With each loop iteration, we will add each record to the loop. We will also create a helper method to generate the marker data:

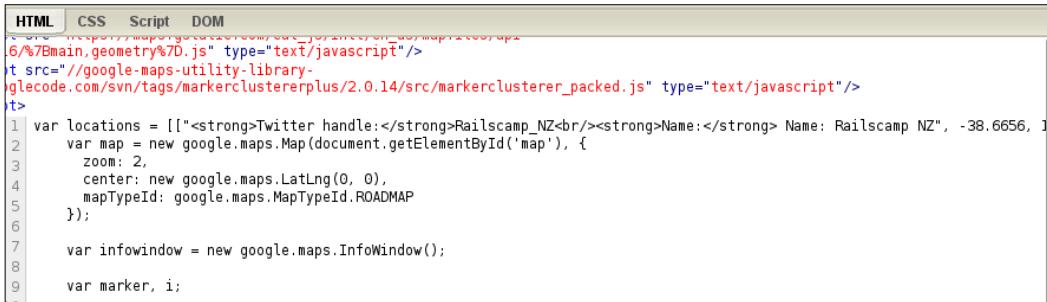
```
app/helpers/map_display_helper.rb
module MapDisplayHelper
  def get_marker_data(screen_name, name) "<strong>Twit:</
strong>#{screen_name}<br/><strong>Name:</strong> Name: #{name}"
  end
end
```

```
app/controllers/map_display_controller.rb
class MapDisplayController < ApplicationController
  def index
    @markers = []
    @friends = current_user.friends
    @friends.each do |f|
      marker_data = get_marker_data(f.screen_name, f.name)
      @markers << [marker_data, f.latitude, f.longitude]
    end
  end
end
```

5. Our final data will be an array of arrays with three keys; data to be displayed in the information box of the marker, the latitude, and the longitude:

```
[["<strong>Twitter Handle:...ong> Name: John Doe1", -43.5321, 172.636], ["<strong>Twitter Handle:...ong> Name: John Doe2", -38.6656, 178.034], ["<strong>Twitter Handle:...g> Name: John Doe3", -37.8141, 144.963], [<strong>Twitter Handle:.../strong> Name: John Doe4", 37.7749, -122.419], [<strong>Twitter Handle:...rong> Name: John Doe5", 37.7141, -122.25], [<strong>Twitter Handle:...rong> Name: John Doe6", 23.6978, 120.961], [<strong>Twitter Handle:...> Name: John Doe7", 19.076, 72.8777], [<strong>Twitter Handle:.../strong> Name: John Doe8", 30.2301, -93.0122], [<strong>Twitter Handle:...me: John Doe9", 22.3964, 114.109], [<strong>Twitter Handle:... Name: John Doe10", 40.7124, -74.0087], [<strong>Twitter Handle:...trong> Name: John Doe11", 37.7749, -122.419], [<strong>Twitter Handle:...g> Name: John Doe12", 52.52, 13.405], [<strong>Twitter Handle:...e: John Doe13", 35.6528, -97.4781], [<strong>Twitter Handle:...trong> Name: John Doe14", 39.9626, -76.7277], [<strong>Twitter Handle:...rong> Name: John Doe15", 32.2617, 76.3068], [<strong>Twitter Handle:...strong> Name: John Doe16", -37.8141, 144.963]]
```

6. The following screenshot shows the preceding data where the Firebug extension of the Chrome browser is used. Check the location variable:



The screenshot shows the Firebug extension's interface with the "Script" tab selected. The code area contains the following JavaScript:

```
6/%7Bmain,geometry%7D.js" type="text/javascript"/>
<t src="//google-maps-utility-library-
glecode.com/svn/tags/markerclustererplus/2.0.14/src/markerclusterer_packed.js" type="text/javascript"/>
<t>
1 var locations = [{"<strong>Twitter handle:</strong>RailsCamp_NZ<br/><strong>Name:</strong> Name: Railscamp NZ", -38.6656, 1
2 var map = new google.maps.Map(document.getElementById('map'), {
3   zoom: 2,
4   center: new google.maps.LatLng(0, 0),
5   mapTypeId: google.maps.MapTypeId.ROADMAP
6 });
7 var infowindow = new google.maps.InfoWindow();
8 var marker, i;
```

Objective complete – mini debriefing

Google Maps requires the marker data to be sent as a hash. The JavaScript reads and understands the data in a particular format. We collected the data we've stored in the database and created a hash such that it can be passed directly to Google Maps. Google Maps will treat this data as the array of markers:

```
[marker_data, f.latitude, f.longitude]
```

The first field in the array will be picked up, converted into HTML, and used for the information box. We added the HTML containing the Twitter handle and username to display the data properly in the Google Maps information box. Then, we added the latitude and longitude of the user's friend. The second and third fields are the latitude and longitude on which the marker is supposed to be pinned and centered. We will display these markers on the map now.

Displaying friends on the map using the Google API

We now have the data in the format that is ready for the Google map. We will use Google Maps v3, the JavaScript API, in order to generate the map and display the markers. We will use the `gmaps4rails` gem but to a very limited capacity. We could use it to generate the entire map. However, considering our scenario, the JavaScript API looks like a better choice. So, we will use the `gmaps4rails` gem to load the basic JavaScript of Google Maps in the asset pipeline.

Engage thrusters

In the following steps, we will create a map and display our friend's data on it using the Google Maps JavaScript API:

1. Add the `gmaps4rails` gem to the `Gemfile` and run `bundle install`:

```
Gemfile
gem 'gmaps4rails', :github => 'apneadiving/Google-Maps-for-Rails'
```

2. We will then load the Google Maps v3 JavaScript in our asset pipeline:

```
app/assets/javascripts/application.js
//= require jquery
//= require jquery_ujs
//= require twitter/bootstrap
//= require underscore
//= require gmaps/google
//= require turbolinks
//= require_tree
```



Google Maps for the Rails JavaScript has been rewritten using CoffeeScript and depends on `underscore.js`. Hence, it is essential to load `underscore.js` as a dependency.

We will also load the necessary dependencies in our view file:

```
app/views/map_display/index.html.erb
<script src="//maps.google.com/maps/api/js?v=3.13&sensor=false
&libraries=geometry" type="text/javascript"></script>
<script src='//google-maps-utility-library-v3.googlecode.com/svn/
tags/markerclustererplus/2.0.14/src/markerclusterer_packed.js'
type='text/javascript'></script>
```

3. We need these two files in order to call the `geometry.js` and Google Maps JavaScript API and Google Maps utility to generate a marker cluster on the map.
4. In order to call the markers data we generated in our previous task, we will initiate a variable in the JavaScript:

```
app/views/map_display/index.html.erb
<script>
    var locations = <%=raw @markers %>;
</script>
```

5. Initiate a script and bind it to an element with the ID `map`. We will tie this to a `div` element. We will center the `map` at $(0, 0)$, that is, at the center of the world:

```
app/views/map_display/index.html.erb
<script>
    var locations = <%=raw @markers %>;
    var map = new google.maps.Map(document.getElementById('map'), {
        zoom: 2,
        center: new google.maps.LatLng(0, 0),
        mapTypeId: google.maps.MapTypeId.ROADMAP
    });
</script>
```

6. We will now create a marker and assign our latitude and longitude values to the marker in a loop. We will also set the content for the information window on each marker:

```
app/views/map_display/index.html.erb
<script>
var marker, i;
for (i = 0; i < locations.length; i++) {
    marker = new google.maps.Marker({
        position: new google.maps.LatLng(locations[i][1],
locations[i][2]),
        map: map
    });
    google.maps.event.addListener(marker, 'click',
(function(marker, i) {
    return function() {
```

```
        infowindow.setContent(locations[i][0]);
        infowindow.open(map, marker);
    }
}) (marker, i));
}
</script>
```

7. The final script with all the dependencies looks like the following code:

```
app/views/map_display/index.html.erb
<script src="//maps.google.com/maps/api/js?v=3.13&sensor=false
&libraries=geometry" type="text/javascript"></script>

<script src='//google-maps-utility-library-v3.googlecode.com/svn/
tags/markerclustererplus/2.0.14/src/markerclusterer_packed.js'
type='text/javascript'></script>
<script>
    var locations = <%= raw @markers %>;
    var map = new google.maps.Map(document.getElementById('map'),
{
    zoom: 2,
    center: new google.maps.LatLng(0, 0),
    mapTypeId: google.maps.MapTypeId.ROADMAP
});
    var infowindow = new google.maps.InfoWindow();
    var marker, i;
    for (i = 0; i < locations.length; i++) {
        marker = new google.maps.Marker({
            position: new google.maps.LatLng(locations[i][1],
locations[i][2]),
            map: map
        });
        google.maps.event.addListener(marker, 'click',
(function(marker, i) {
            return function() {
                infowindow.setContent(locations[i][0]);
                infowindow.open(map, marker);
            }
        }) (marker, i));
    }
</script>
```

8. Finally, we will display the map in the `div` element. We will bind the JavaScript to the `div` element using the ID name `map`:

```
app/views/map_display/index.html.erb
<div id="map" style='width: 1200px; height: 600px;'></div>
```

Objective complete – mini debriefing

In this task, we created the JavaScript for generating the Google map and plotting all the data in the form of markers for us. We used the `gmap4rails` gem to load the Google Maps JavaScript API into our asset pipeline. Google Maps for Rails is wrapped on top of the Google Maps JavaScript API. It is completely rewritten in Coffee and `underscore.js`. `Underscore.js` is a library that provides a set of specialized functional programming helper methods. Some of the methods that Google Maps for Rails uses are as follows:

- ▶ `_.extend`
- ▶ `_.map`
- ▶ `_.isFunction`
- ▶ `_.each`
- ▶ `_.isObject`

Then, we defined some geometry and marker-specific JavaScript in our views. We initiated a map and associated it with an element with the `map` ID. Then, the loop will read the collection of marker data represented as `@marker` variable in our `map_display_controller` and call the location coordinates from there. The `locations[i][1]` and `locations[i][2]`, the second and the third element of the `locations` array are called as the collection is looped over:

```
var marker, i;
for (i = 0; i < locations.length; i++) {
    marker = new google.maps.Marker({
        position: new google.maps.LatLng(locations[i][1], locations[i]
[2]),
        map: map
    });
}
```

Then, we will pass the first value of the array to the information window on the map and bind it to the `click` event:

```
google.maps.event.addListener(marker, 'click', (function(marker, i) {
    return function() {
        infowindow.setContent(locations[i][0]);
        infowindow.open(map, marker);
    }
})(marker, i));
}
```

We used the `raw` tag to pass the marker data to the Google Maps JavaScript. By default, Rails escapes the executable script within the objects:

```
<%=raw @markers %>
```

The `raw` tag is equivalent to the `html_safe` tag in Rails. The difference lies in how they handle the `nil` object. The `html_safe` tag gives an exception, whereas `raw` gives out an empty string in return. Also, in some cases `raw` is susceptible to an XSS attack. This can be a use case where we have a text area and the attacker inserts an executable JavaScript in it. We should avoid the use of `raw` in those cases. In our case, `raw` is handled to output the data from a hash that we build. The marker data is now being displayed in blurb on the marker as shown in the following screenshot:



Creating points of interest – filter users based on their location

Grouping similar information on the map according to a specific criteria is called points of interest. This is a term used for markers or points on the maps that can be categorized or grouped together. We will use locations as the points of interest in our application. We will call all the locations in our system and search the friends according to it. We will use the geocoder API to do this.

Engage thrusters

We will create location-based filters for our users in this task:

1. The geocoder gem has a method called `near`, which takes the location string as the parameter and runs a spatial query on the database:

```
1.9.3p327 :001 > user = User.first
1.9.3p327 :002 > user.friends.near("NY")
```

```
Loading development environment (Rails 4.1.0.rc1)
1.9.3p327 :001 > user = User.first
  User Load (0.7ms)  SELECT `users`.* FROM `users` ORDER BY `users`.`id` ASC LIMIT 1
=> #<User id: 1, provider: "twitter", uid: "415386785", name: "Saurabh Bhatia", oauth_token: "415386785-URRXAJQSyyJlFkJQt2eSyg4phXajoAjPxVpzUPI", oauth_expires_at: nil, created_at: "2013-12-07 09:11:49", updated_at: "2013-12-07 09:11:49", address: "Zhubei City, Taiwan", a
vatar: "http://pbs.twimg.com/profile_images/3408461966/ec4...", oauth_secret: "0ix2ddTld19Vj8i5NyCxX6gtqlXu6NY14RFSXywDZJD", latitude:
"121.019444", longitude: nil>
1.9.3p327 :002 > user.friends.near("NY")
  Friend Load (48.8ms)  SELECT friends.* ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2), 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.00597309999999 - friends.longitude) * PI() / 180 / 2), 2))) AS distance, CAST(DEGREES(ATAN2(RADIANS(friends.longitude - 40.7143528)) + 360 AS decimal) % 360 AS bearing FROM `friends` WHERE `friends`.`user_id` = 1 AND (friends.latitude BETWEEN 40.4248892337783 AND 41.0038163662217 AND friends.longitude BETWEEN -74.38786578682475 AND -73.62408041317524 AND 3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2), 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.00597309999999 - friends.longitude) * PI() / 180 / 2), 2))) <= 20) ORDER BY distance ASC
=> #<ActiveRecord::AssociationRelation:0x007fbc401000000 @klass=User, @name="friends", @options={:order=>{:distance=>:asc}}, @primary_key="id", @table="friends", @type=:has_many>
  => #<Friend id: 63, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:27:46", updated_at: "2014-02-28 05:27:46", #<Friend id: 80, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:28:01", updated_at: "2014-02-28 05:28:01", latitude: 40.7144, longitude: -74.006>, #<Friend id: 97, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:45:17", updated_at: "2014-02-28 05:45:17", latitude: 40.7144, longitude: -74.006>, #<Friend id: 73, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:27:47", updated_at: "2014-02-28 05:27:47", latitude: 40.7124, longitude: -74.0087>, #<Friend id: 90, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:28:03", updated_at: "2014-02-28 05:28:03", latitude: 40.7124, longitude: -74.0087>, #<Friend id: 107, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:45:19", updated_at: "2014-02-28 05:45:19", latitude: 40.7124, longitude: -74.0087>]
1.9.3p327 :003 > ■
```

2. We got six results when we searched for the term NY:

```
1.9.3p327 :003 > user.friends.near("NY").length
  Friend Load (1.6ms)  SELECT friends.* ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.00597309999999 - friends.longitude) * PI() / 180 / 2), 2))) AS distance, CAST(DEGREES(ATAN2(RADIANS(friends.longitude - 40.7143528)) + 360 AS decimal) % 360 AS bearing FROM `friends` WHERE `friends`.`user_id` = 1 AND (friends.latitude BETWEEN 40.4248892337783 AND 41.0038163662217 AND friends.longitude BETWEEN -74.38786578682475 AND -73.62408041317524 AND 3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2), 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.00597309999999 - friends.longitude) * PI() / 180 / 2), 2))) <= 20) ORDER BY distance ASC
```

3. Now, as we know that we can find people based on their locations, we will add the location of the friend as a parameter called `place` in our query:

```
app/controllers/map_display_controller.rb
if params[:place].present?
  @friends = current_user.friends.near(params[:place])
  @friends.each do |f|
    end
end
```

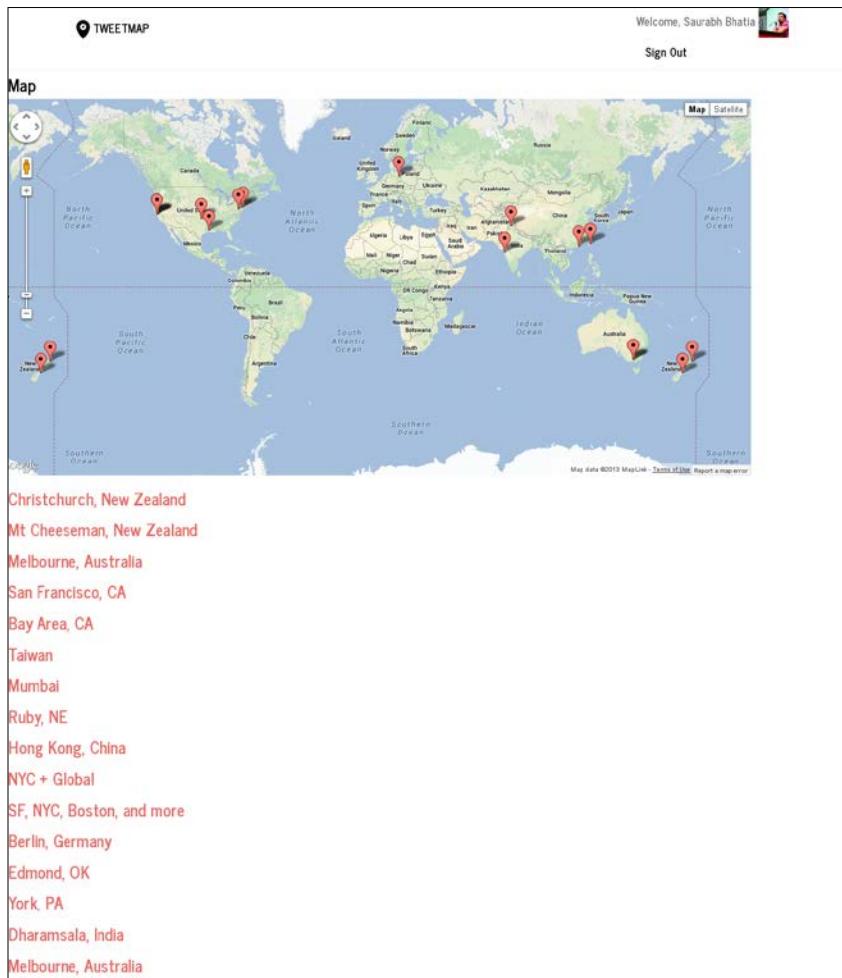
4. So, our final method looks like the following code:

```
app/controllers/map_display_controller.rb
class MapDisplayController < ApplicationController
  include MapDisplayHelper
  def index
    @markers = []
    if params[:place].present?
      @friends = current_user.friends.near(params[:place])
      @friends.each do |f|
        marker_data = get_marker_data(f.screen_name, f.name)
        @markers << [marker_data, f.latitude, f.longitude]
      end
    else
      @friends = current_user.friends
      @friends.each do |f|
        marker_data = get_marker_data(f.screen_name, f.name)
        @markers << [marker_data, f.latitude, f.longitude]
      end
    end
  end
end
```

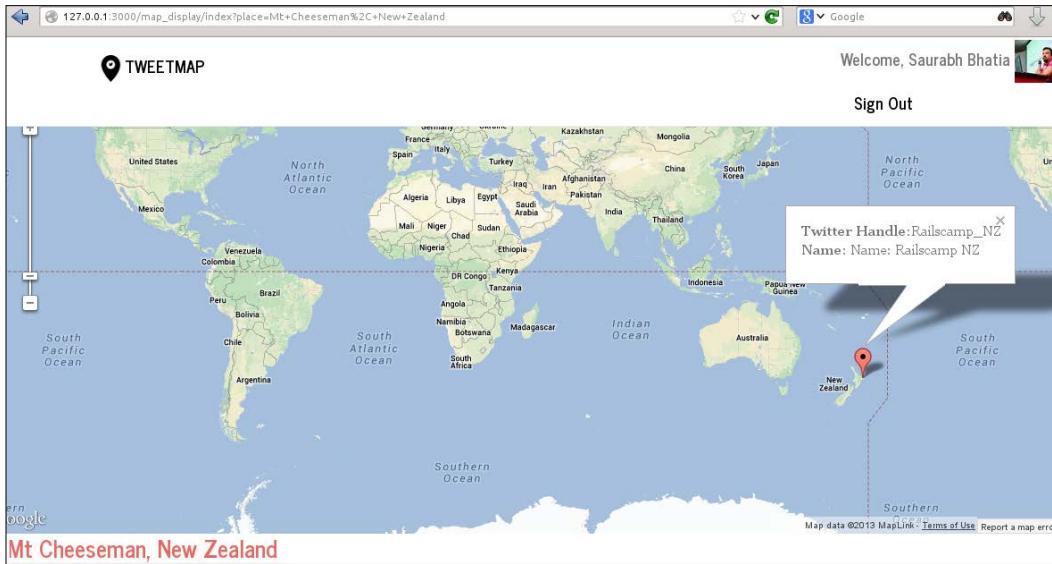
5. In our view, we need to pass the `place` parameter to the controller method:

```
app/views/map_display/index.html.erb
<div class="col-lg-12"><% @friends.each do |f| %><h3 id="type-blockquotes"><%= link_to "#{f.location}", map_display_index_path(:place => "#{f.location}") %></h3><% end %></div>
```

6. The loop will generate a list of location links with the `place` parameter. The following screenshot displays the list of places as links:



7. When we click on one of the preceding locations, we will get a map with the filtered values and the location values:



Objective complete – mini debriefing

In the previous task, we used the geocoder API to search our database. It extends spatial queries in the database to search the `friends` table according to the location coordinates. The `near` query first finds the coordinates of the location according to which it is searched. Then, it converts the latitude and longitude to degrees and radians in order to match it to the location coordinates of the friends. The following is an example of geo query we can run using the geocoder:

```
Loading development environment (Rails 4.1.0.rc1)
1.9.3p327 :001 > Friend.near("NY")
  Friend Load (49.1ms)  SELECT `friends`.*, 3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2), 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.00597309999999 - friends.longitude) * PI() / 180 / 2), 2))) AS distance, CAST(DEGREES(ATAN2( RADIANS(friends.longitude - -74.00597309999999), RADIANS(friends.latitude - 40.7143528))) + 360 AS decimal) % 360 AS bearing FROM `friends` WHERE (friends.latitude BETWEEN 40.4248892337783 AND 41.0038163662217 AND friends.longitude BETWEEN -74.3878657882475 AND -73.62408041317524 AND 3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((40.7143528 - friends.latitude) * PI() / 180 / 2), 2) + COS(40.7143528 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((-74.0059730999999 - friends.longitude) * PI() / 180 / 2), 2))) <= 20) ORDER BY distance ASC
=> #<ActiveRecord::Relation:0x007fbc40000000>
  => #<Friend id: 63, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:27:46", updated_at: "2014-02-28 05:27:46", #<Friend id: 80, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:28:01", updated_at: "2014-02-28 05:28:01", #<Friend id: 97, name: "Joel Spolsky", screen_name: "spolsky", location: "New York, NY", user_id: 1, created_at: "2014-02-28 05:45:17", updated_at: "2014-02-28 05:45:17", #<Friend id: 73, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:27:47", updated_at: "2014-02-28 05:27:47", #<Friend id: 90, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:45:19", updated_at: "2014-02-28 05:45:19", #<Friend id: 107, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:45:19", updated_at: "2014-02-28 05:45:19">]
1.9.3p327 :002 > ■
```

Creating an API Mashup – Twitter and Google Maps

The geocoder API provides a host of other features, such as finding the distance between two friends and the `nearbys` query. The `nearbys` query is run as follows:

```
Friend.last.nearbys(30)
```

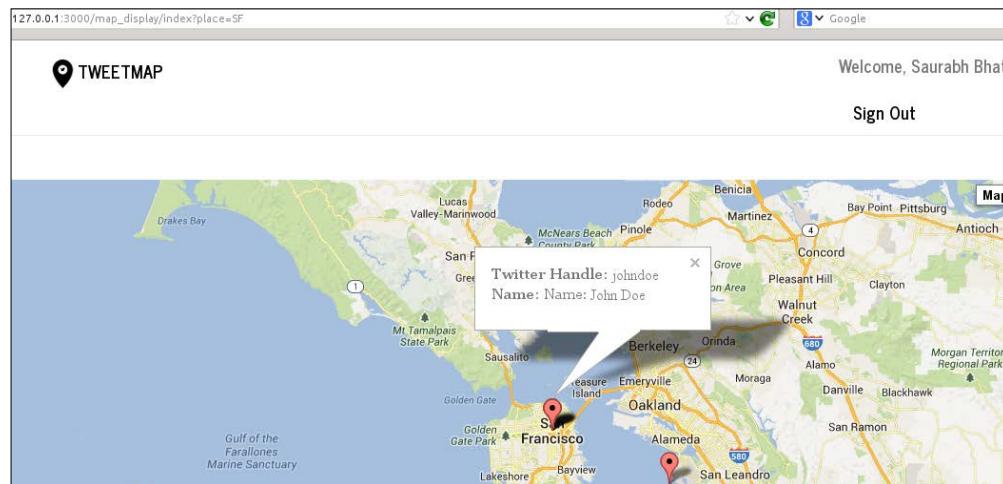
```
Loading development environment (Rails 4.1.0.rc1)
1.9.3p327 :001 > Friend.last.nearbys(30)
  Friend Load (0.5ms)  SELECT `friends`.* FROM `friends` ORDER BY `friends`.`id` DESC LIMIT 1
  Friend Load (1.0ms)  SELECT `friends`.* FROM `friends` ORDER BY `friends`.`id` ASC LIMIT 1
    => #<ActiveRecord::Relation:0x007f8000000000>
      3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((-37.8141 - friends.latitude) * PI() / 180 / 2), 2) + C
OS(-37.8141 * PI()) / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((144.963 - friends.longitude) * PI() / 180 / 2), 2))) AS distance, CAST(DEGREES(ATAN2(RADIANS(friends.longitude - 144.963), RADIANS(friends.latitude - -37.8141))) + 360 AS decimal) % 360 AS bearing
      FROM `friends` WHERE (friends.latitude BETWEEN -38.248295349332544 AND -37.37990465066746 AND friends.longitude BETWEEN 144.41338852602124 AND 145.51261147397875 AND 3958.755864232 * 2 * ASIN(SQRT(POWER(SIN((-37.8141 - friends.latitude) * PI() / 180 / 2), 2) + COS(-37.8141 * PI() / 180) * COS(friends.latitude * PI() / 180) * POWER(SIN((144.963 - friends.longitude) * PI() / 180 / 2), 2))) <= 30 AND friends.id != 113) ORDER BY distance ASC
      => #<ActiveRecord::Relation:0x007f8000000000>
        66, name: "SitePoint Ruby", screen_name: "SitePointRuby", location: "Melbourne, Australia", user_id: 1, created_at: "2014-02-28 05:27:46", latitude: -37.8141, longitude: 144.963, #<Friend id: 79, name: "Kind Of A Bigg Dill", screen_name: "ryanbigg", location: "Melbourne, Australia", user_id: 1, created_at: "2014-02-28 05:27:48", updated_at: "2014-02-28 05:27:48", latitude: -37.8141, longitude: 144.963, #<Friend id: 83, name: "SitePoint Ruby", screen_name: "SitePointRuby", location: "Melbourne, Australia", user_id: 1, created_at: "2014-02-28 05:28:02", latitude: -37.8141, longitude: 144.963, #<Friend id: 96, name: "Kind Of A Bigg Dill", screen_name: "ryanbigg", location: "Melbourne, Australia", user_id: 1, created_at: "2014-02-28 05:28:04", updated_at: "2014-02-28 05:28:04", latitude: -37.8141, longitude: 144.963, #<Friend id: 100, name: "SitePoint Ruby", screen_name: "SitePointRuby", location: "Melbourne, Australia", user_id: 1, created_at: "2014-02-28 05:45:17", updated_at: "2014-02-28 05:45:17", latitude: -37.8141, longitude: 144.963>]
1.9.3p327 :002 > ■
```

We can also find distance in miles between two friends. The results are as shown in the following screenshot:

```
Geocoder::Calculations.distance_between(friend1, friend2)
```

```
Loading development environment (Rails 4.1.0.rc1)
1.9.3p327 :001 > friend1 = Friend.first
  Friend Load (0.6ms)  SELECT `friends`.* FROM `friends` ORDER BY `friends`.`id` ASC LIMIT 1
  => #<Friend id: 65, name: "RailsCamp NZ", screen_name: "RailsCamp_NZ", location: "Mt Cheeseman, New Zealand", user_id: 1, created_at: "2014-02-28 05:27:46", updated_at: "2014-02-28 05:27:46", latitude: -38.6656, longitude: 178.034>
1.9.3p327 :002 > friend2 = Friend.last
  Friend Load (0.9ms)  SELECT `friends`.* FROM `friends` ORDER BY `friends`.`id` DESC LIMIT 1
  => #<Friend id: 107, name: "General Assembly", screen_name: "GA", location: "NYC + Global", user_id: 1, created_at: "2014-02-28 05:45:19", updated_at: "2014-02-28 05:45:19", latitude: 40.7124, longitude: -74.0087>
1.9.3p327 :003 > Geocoder::Calculations.distance_between(friend1, friend2)
=> 8716.54505306303
1.9.3p327 :004 > ■
```

The following screenshot shows the result of a `near` query when clicked on San Francisco:



Mission accomplished

We have successfully created a fun app that will map our friends on Twitter. We can broadly divide what we did into four parts:

- ▶ Used OmniAuth to sign up and log in with Twitter
- ▶ Created sessions with Twitter and maintained our user in the session
- ▶ Used Twitter v1.1 API and the `twitter` gem to call the users and their friends' data
- ▶ Used the Ruby geocoding API to find the location coordinates of each friend and used mapping to display these friends on the Google map using Google Maps v3

Hotshot challenges

We can still have a lot of fun with these APIs:

- ▶ Display the last tweet of each user in the information window
- ▶ Display the user avatar in the information window
- ▶ Change the location filter to a checkbox
- ▶ Use jQuery to send the parameters to the controller
- ▶ Find the distance between two friends

Project 8

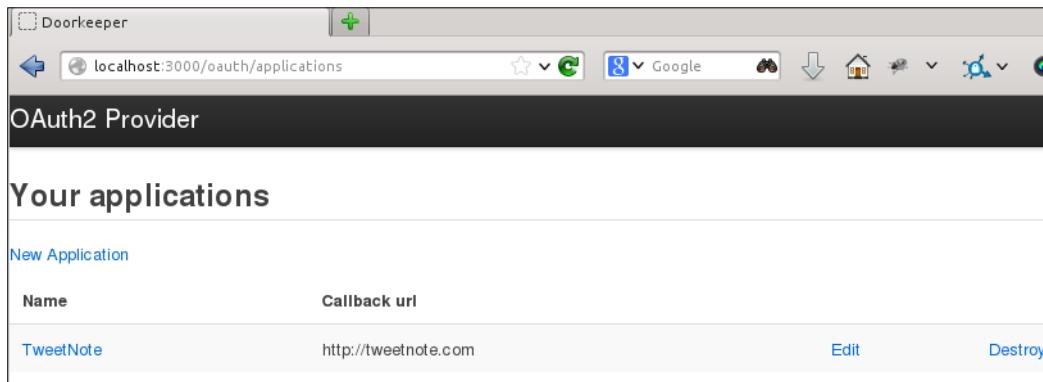
API Only Application – Backend for a Mobile App

Android and iOS have taken smartphones to a new level of sophistication, and their respective application ecosystems have created a huge movement among developers. In order to do so, a lot of these applications need background processing, data storage, and data manipulation, along with authentication and authorization. In such cases, we need an API only application, where the application handles the processing load and returns data to a mobile client using an API.

Mission briefing

In an API only application, only models and controllers exist. This kind of application provides an interface to the data, apart from a full-blown application with business logic in the backend. The application can serve as a backend to a mobile application. The application only behaves like an API, with no UI layer present for the data. We will create an application to make notes and send data to an application using a REST client. As we create an API, we will also see how to create an OAuth2 provider to authorize applications with it.

The following screenshot shows how our application's OAuth provider screen will look at the end of our project:



A screenshot of a web browser window titled "Doorkeeper". The address bar shows "localhost:3000/oauth/applications". The main content area is titled "OAuth2 Provider" and "Your applications". It lists a single application: "TweetNote" with "http://tweetnote.com" as the callback URL. There are "Edit" and "Destroy" links next to each row.

Name	Callback url	Edit	Destroy
TweetNote	http://tweetnote.com	Edit	Destroy

Why is it awesome?

Rails has a subproject called Rails API that takes a leaner approach to the creation of these applications. It strips off a lot of middleware components and the view layer out of the application. We can conditionally add some middleware and controller modules wherever required. We will also convert our application into the OAuth provider so that the authentication on our mobile applications happens via OAuth. We will use the `doorkeeper` gem to create an OAuth application ID and OAuth secret, along with a callback URL.

At the end of this project, we will be able to create an API only application with OAuth end points to authorize applications for the data API.

Your Hotshot objectives

While building this application, we will have to go through the following tasks:

- ▶ Creating, editing, and deleting notes
- ▶ Arranging notes category wise
- ▶ Sending join and association data via JSON
- ▶ Creating an OAuth2 provider
- ▶ Generating API keys
- ▶ Securing the application

Mission checklist

We need to install the following on the system and sign up for the API keys before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0.0
- ▶ MySQL
- ▶ Rails API
- ▶ Devise
- ▶ Git
- ▶ Doorkeeper
- ▶ jQuery

Creating, editing, and deleting notes

Our first task involves certain tasks that have already been done in the previous projects, but now we are going to do it in a slightly different way. As the main aim of our application is to be leaner and faster than a normal application, we will add some rack, action controller, and action view modules only when required. Some middleware stack (rack modules) and view-related stack (assets and views) components are stripped off. Please be sure to go through the Readme section of Rails API in detail (<https://github.com/rails-api/rails-api/blob/master/README.md>). It contains a list of modules that are included in a default Rails API application and what they are used for. It also has a list of modules that can be included in order to extend the default stack as and when required.

Engage thrusters

We will start by installing Rails API and generating our skeleton application by performing the following steps:

1. We will install Rails API first using the following command:
`$ gem install rails-api`
2. Once the gem is installed, we will generate a blank application using the gem. We will use MySQL to develop our application as follows:

```
rails-api new notely -d mysql
create
create  README.rdoc
create  Rakefile
```

```
create config.ru
create .gitignore
create Gemfile
create app
create app/controllers/application_controller.rb
create app/assets/images/.keep
create app/mailers/.keep
create app/models/.keep
create app/controllers/concerns/.keep
create app/models/concerns/.keep
create bin
create bin/bundle
create bin/rails
create bin/rake
create config
create config/routes.rb
create config/application.rb
create config/environment.rb
create config/environments
create config/environments/development.rb
create config/environments/production.rb
create config/environments/test.rb
create config/initializers
create config/initializers/secret_token.rb
create config/initializers/wrap_parameters.rb
create config/locales
create config/locales/en.yml
create config/boot.rb
create config/database.yml
create db
create db/seeds.rb
create lib
create lib/tasks
create lib/tasks/.keep
create lib/assets
create lib/assets/.keep
```

```
create  log
create  log/.keep
create  public
create  public/404.html
create  public/422.html
create  public/500.html
create  public/favicon.ico
create  public/robots.txt
create  test/fixtures
create  test/fixtures/.keep
create  test/controllers
create  test/controllers/.keep
create  test/mailers
create  test/mailers/.keep
create  test/models
create  test/models/.keep
create  test/helpers
create  test/helpers/.keep
create  test/integration
create  test/integration/.keep
create  test/test_helper.rb

run  bundle install
Fetching gem metadata from https://rubygems.org/
Fetching gem metadata from https://rubygems.org/..
Resolving dependencies...
Using rake (10.1.0)
Using i18n (0.6.9)
Using minitest (4.7.5)
Using multi_json (1.8.2)
Using atomic (1.1.14)
Using thread_safe (0.1.3)
Using tzinfo (0.3.38)
Using activesupport (4.0.1)
Using builder (3.1.4)
Using erubis (2.7.0)
Using rack (1.5.2)
```

```
Using rack-test (0.6.2)
Using actionpack (4.0.1)
Using mime-types (1.25.1)
Using polyglot (0.3.3)
Using treetop (1.4.15)
Using mail (2.5.4)
Using actionmailer (4.0.1)
Using activemodel (4.0.1)
Using activerecord-deprecated_finders (1.0.3)
Using arel (4.0.1)
Using activerecord (4.0.1)
Using bundler (1.3.5)
Using hike (1.2.3)
Using mysql2 (0.3.14)
Using thor (0.18.1)
Using railties (4.0.1)
Using tilt (1.4.1)
Using sprockets (2.10.1)
Using sprockets-rails (2.0.1)
Using rails (4.0.1)
Using rails-api (0.1.0)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is
installed.
```

3. Rails API is under constant development, and its compatibility with Rails 4 is being improved day by day. In order to avoid some bugs and pitfalls, we will bundle Rails API from the master as follows:

Gemfile

```
gem 'rails', '4.0.1'
gem 'rails-api', git: 'https://github.com/rails-api/rails-api.
git', branch: 'master'
```

4. Set up database.yml according to your local machine credentials.

5. Now that our Rails API base project is set up, we will generate a model and a controller to create our notes as follows:

```
notely$rails g scaffold note title:string body:string
      invoke  active_record
      create    db/migrate/20131211143114_create_notes.rb
      create    app/models/note.rb
      invoke  test_unit
      create    test/models/note_test.rb
      create    test/fixtures/notes.yml
      invoke  api_resource_route
      route    resources :notes, except: [:new, :edit]
      invoke  scaffold_controller
      create    app/controllers/notes_controller.rb
      invoke  test_unit
      create    test/controllers/notes_controller_test.rb
```

6. We will add a very basic version for our API. In order to do this, we will create a namespace in routes:

```
config/routes.rb
```

```
Notely::Application.routes.draw do
  namespace :api do
    namespace :v1 do
      resources :notes
    end
  end
end
```

7. This will give an error because we need to create the same namespace in the controllers folder and move our notes_controller.rb file to api/v1 as follows:

```
notely/app/controllers$mkdir api
notely/app/controllers$ cd api
notely/app/controllers/api$mkdir v1
notely/app/controllers/$cd ..
notely/app/controllers/$mv notes_controller.rb api/v1
```

8. Our application_controller.rb file extends from ActionController::API as follows:

```
app/controllers/application_controller.rb
class ApplicationController < ActionController::API
end
```

9. Now that we have created a namespace, we also need to convert our controller to work with this namespace. We will also refactor the code to use before_action in order to set the note id for multiple actions as follows:

```
app/controllers/api/v1/notes_controller.rb
#namespace for api
module Api
  module V1
    class NotesController < ApplicationController

      def index
        @notes = Note.all
        render json: @notes
      end

      def new
        @part = Part.new
      end

      def show
        @note = Note.find(params[:id])

        render json: @note
      end

      def create
        @note = Note.new(params[:id])
        if @note.save
          render json: @note, status: :created, location:
            @note
        end
      end
    end
  end
end
```

```
    else
      render json: @note.errors, status:
        :unprocessable_entity
    end
  end

  def update

    @note = Note.find(params[:id])

    if @note.update(params[:id])
      head :no_content
    else
      render json: @note.errors, status:
        :unprocessable_entity
    end
  end

  def destroy

    @note.destroy
    head :no_content
  end

end
end
end
```

10. We have to make Rails API and Rails 4 compatible with strong parameters because we need to post data to the API and pass the data as parameters in our client. We will first create an initializer to load a `StrongParameters` module along with the `ActionController::API` module as follows:

```
config/initializers/strong_parameters.rb
ActionController::API.send :include, ActionController::StrongParameters
```

11. We also need to modify our controller to the Rails 4 format by passing params through a private method. The default controller generated by Rails API is not in accordance with the Rails 4 format to accept the parameters:

```
app/controllers/api/v1/notes_controller.rb

private
  def set_note
    @note = Note.find(params[:id])
  end

  # Never trust parameters from the scary internet, only allow the
  # white list through.
  def note_params
    params.require(:note).permit(:title,
      :body, :category_id) if params[:note]
  end
```

12. To use these params, we need to modify the create and update methods to accept parameters through the note_params variable:

```
app/controllers/api/v1/notes_controller.rb

# POST /notes
# POST /notes.json
def create
  @note = Note.new(note_params)

  if @note.save
    render json: @note, status: :created, location: @note
  else
    render json: @note.errors, status:
      :unprocessable_entity
  end
end

# PATCH/PUT /notes/1
# PATCH/PUT /notes/1.json
def update
```

```
if @note.update(note_params)
  head :no_content
else
  render json: @note.errors, status:
  :unprocessable_entity
end
end
```

13. We will add a module in order to serialize the JSON data input and output of our API. It adds a more object-oriented approach as opposed to a hash-oriented approach to JSON:

```
Gemfile
gem "active_model_serializers"

notely$bundle install
```

14. We already have a model before adding the serializer, hence we will run the following command to generate a serializer for our existing model:

```
notely$ rails g serializer note
create app/serializers/note_serializer.rb
```

15. We will modify our serializer in order to accept all the attributes:

```
app/serializers/note_serializer.rb

class NoteSerializer < ActiveModel::Serializer
  attributes :id, :title, :body
end
```

Objective complete – mini debriefing

In this task, we generated a Rails API project with Rails 4.1 and MySQL as the database. We can use any database we want, such as PostgreSQL and MongoDB. However, we chose MySQL for the sake of simplicity of demonstration. It would be noteworthy to know that in the application generated by Rails API, the application controller inherits from `ActionController::API` instead of the `ActionController::base` class:

```
class ApplicationController < ActionController::API
end
```

We also developed a skeleton to create the notes. We created a very basic API with this version. We bridged the gap between Rails 4 and Rails API to accept the parameters.

We used the active serializer module in order to serialize JSON, generating a serializer for the existing model. We added attributes for `note` to the serializer using the following method:

```
def note_params
  params.require(:note).permit(:title, :body, :category_id) if
  params[:note]
end
```

There are several JSON template options such as JBuilder and Rabl to name a couple. JBuilder comes by default with Rails, whereas Rabl can be added as a gem. One of the primary reasons to select the `ActiveModel` serializer over the other two is performance. As the dataset increases in size, the process of JSON generation slows down, thus affecting the overall performance of the API. The `ActiveModel` serializer, however, has been known to perform better than these libraries.

In the preceding method, we have enabled the posting of params only if they are present. We also added a serializer to package this JSON hash as an object and posted it to the `params` method. This attribute definition will make `params` an attribute of an object called `note`, which is posted to `params[:note]`:

```
class NoteSerializer < ActiveModel::Serializer
  attributes :id, :title, :body
```

```
end
```

When we visit `notes.json`, we will see the following code snippet:

```
{"notes": [
  [
    {"id":1,"title":"First Note","body":"Buy a new ram"},  

    {"id":2,"title":"Second Note","body":"Buy Macbook pro"}
  ]
}]
```

The complete URL looks like `http://localhost:3000/api/v1/notes.json`. Since we have created a controller namespace and added a version, we created versioned endpoints for our API. When we write Version 2, we can simply create a new namespace for v2 with the same controllers, methods, and updated code. In that way, we can keep both versions of the API live parallelly:

```
namespace :api do
  namespace :v1 do
    resources :notes
```

```
end  
end  
resources :notes
```

In order to make this namespace work in the controller, we need to define our controller like a module. By default, Rails treats the path elements as modules.

From time to time, we are expected to make major changes in our application. As and when our application is updated, there are several changes it goes through, such as the structure of data, changes in fields, and so on. Sometimes the changes are not backward compatible and there are clients already using our application. In order for them to keep using our application effectively, we need a new version of the API, rather than changing the entire API entity itself.

Arranging notes category wise

In our application, users will need to arrange their notes category wise. A category will act as a search filter to keep the notes. We will add an association via the models. This task deals with the creation of the category model and its association with the note model.

Engage thrusters

We will now add categories to our note application:

1. We will first create a category model for our application using the following command:

```
notely$ rails g model category title:string
```

2. The migration file generated looks as follows:

```
class CreateCategories < ActiveRecord::Migration  
  def change  
    create_table :categories do |t|  
      t.string :title  
  
      t.timestamps  
    end  
  end  
end  
  
:~/notely$bundle exec rake db:migrate
```

3. We will save the category title for now and we will try to add categories using our Rails console:

```
:~/notely$ rails c
Loading development environment (Rails 4.0.1)
1.9.3-p327 :001 > category = Category.new
=> #<Category id: nil, title: nil, created_at: nil, updated_at:
nil>
1.9.3-p327 :002 > category.title = "Personal"
=> "Personal"
1.9.3-p327 :003 > category.save
(0.4ms)  BEGIN
SQL (38.1ms)  INSERT INTO `categories` (`created_at`, `title`,
`updated_at`) VALUES ('2013-12-26 00:02:21', 'Personal', '2013-12-
26 00:02:21')
(53.9ms)  COMMIT
=> true
```

4. We will then add a `has_many` relation in the `category` model as follows:

app/models/category.rb

```
class Category < ActiveRecord::Base
  has_many :notes
end
```

5. Likewise, we will add a `belongs_to` relationship in our `note` model:

app/models/note.rb

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

6. In order for the association to work, we will add a `category_id` column to our `note` table:

```
notely$ rails g migration add_category_id_to_notes category_
id:integer
      invoke active_record
      create   db/migrate/20131226000927_add_category_id_to_
notes.rb
```

7. The migration will look as follows:

```
db/migrate$ nano 20131214094532_add_category_id_to_notes.rb
class AddCategoryIdToNotes < ActiveRecord::Migration
  def change
    add_column :notes, :category_id, :integer
  end
endFinally run bundle exec rake db:migratedb:migrate to generate
the table
notely$ bundle exec rake db:migrate
```

8. So, our schema now looks like the following:

```
db/schema.rb
create_table "categories", force: true do |t|
  t.string    "title"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "notes", force: true do |t|
  t.string    "title"
  t.string    "body"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.integer   "category_id"
end
```

9. Now that we have added the `category_id` column, we will also have to update our serializer:

```
app/serializers/note_serializer.rb

class NoteSerializer < ActiveModel::Serializer
  attributes :id, :title, :body, :category_id
end
```

Objective complete – mini debriefing

In the preceding task, we created a `category` model. We created an association between the `category` and `note` models. The association is such that a `category` has many `notes` and a `note` belongs to a `category`. We also created the required migrations and added the respective fields to the database.

Sending join data via JSON

In the previous tasks, we created notes and then associated them with the categories. Now, we will customize our `serializer` class in order to work with the association. We will also use a REST client to see how to get data and post data to our API. Of course, using it in the client is the best way; however, in order to see if the data is being inserted correctly or not, we need a command-line interface. We will use the REST client's command-line interface to interact with our API.

Engage thrusters

We will serialize our data and prepare to get, post, and put the data by performing the following steps:

1. We will first add the association data to our `ActiveModel` serializer. We will add the `has_one :category` method and embed it in order to automatically add `category_id` to our JSON object as follows:

```
app/serializers/note_serializer.rb
class NoteSerializer < ActiveModel::Serializer
  embed :id

  attributes :id, :title, :body
  has_one :category
end
```

2. When we navigate to our notes URL, we get the following JSON values with `category_id` appended to them. An alternate way to check the response is via the `curl` command:

```
http://localhost:3000/api/v1/notes
{
  "notes": [
    {
      "id": 1, "title": "First Note", "body": "Buy a new
      ram", "category_id": 1},
    {
      "id": 2, "title": "Second Note", "body": "Buy Macbook
      pro", "category_id": 2}
  ]
}
```

3. The first step to start testing whether our API works is to install the REST client:

```
$ gem install rest-client
```

4. The REST client is accessible as a command-line tool. So, we will open our interactive Ruby shell and try calling our API:

```
$ irb
1.9.3-p327 :001 > require 'rubygems'
=> false
1.9.3-p327 :002 > require 'rest-client'
=> true
1.9.3-p327 :004 > response = RestClient.get 'http://
localhost:3000/api/v1/notes'
=> "{\"notes\":[{\\"id\\":1,\"title\\\":\"First Note\",\"body\\\":\"Buy
a new ram\"}, {\"category_id\\":1},{\"id\\":2,\"title\\\":\"Second
Note\"}, {\"body\\\":\"Buy Macbook pro\"}, {\"category_id\\":2},{\"id\\":3,
\"title\\\":\"note\"}, {\"body\\\":\"study\"}, {\"category_id\\\":1}]"
=> "
```

5. We stored our response in a variable so that we can see some of the common attributes of the response. We can return the response code and headers for the sake of testing:

```
1.9.3-p327 :005 > response.code
=> 200
1.9.3-p327 :006 > response.headers
=> { :x_frame_options=>"SAMEORIGIN", :x_xss_protection=>"1;
mode=block", :x_content_type_options=>"nosniff", :x_ua_
compatible=>"chrome=1", :content_type=>"application/json;
charset=utf-8", :etag=>"\"852ad43f6964fa588ce190c8fc8c7239\"", 
:cache_control=>"max-age=0, private, must-revalidate", :x_
request_id=>"20bed29f-bffb-4743-9887-f427686c7187", :x_
runtime=>"0.030178", :transfer_encoding=>"chunked"}
```

6. We will try posting our first note using the API:

```
1.9.3-p327 :013 > RestClient.post('http://localhost:3000/api/v1/
notes', { :note => { :title => 'test', :body => 'body', :category_id
=> 2 } })
=>
"{"note":{\"id\":8,\"title\":\"test\",\"body\":\"body\",
\"category_id\":2}"}
```

7. In our server log, we can see the post request and the 201 response code:

```
Started POST "/api/v1/notes" for 127.0.0.1 at 2013-12-27 07:43:04
+0800
Processing by Api::V1::NotesController#create as XML
Parameters: {"note"=>{"title"=>"test", "body"=>"body",
"category_id"=>"2"}}
(0.3ms)  BEGIN
```

```
SQL (2.8ms)  INSERT INTO `notes` (`body`, `category_id`,  
`created_at`, `title`, `updated_at`) VALUES ('body', 2, '2013-12-  
26 23:43:04', 'test', '2013-12-26 23:43:04')  
          (49.0ms)  COMMIT  
Category Load (0.5ms)  SELECT `categories`.* FROM `categories`  
WHERE `categories`.`id` = 2 ORDER BY `categories`.`id` ASC LIMIT 1  
Completed 201 Created in 131ms (Views: 10.3ms | ActiveRecord:  
52.6ms)
```

8. Once we see the 201 response code, we can check if the value has been inserted successfully in our database or not:

```
$ rails c  
Loading development environment (Rails 4.0.1)  
1.9.3-p327 :001 > Note.last  
Note Load (0.8ms)  SELECT `notes`.* FROM `notes` ORDER BY  
`notes`.`id` DESC LIMIT 1  
=> #<Note id: 8, title: "test", body: "body", created_at: "2013-  
12-26 23:43:04", updated_at: "2013-12-26 23:43:04", category_id:  
2>
```

9. We will now check both the index and show methods using our REST client:

```
1.9.3p327 :010 > RestClient.get 'http://localhost:3000/api/v1/  
notes'  
=> "{\"notes\": [{\"id\":1,\"title\":\"First Note\",\"body\":\"Buy  
a new ram\"}, {\"id\":1}, {\"id\":2,\"title\":\"Second  
Note\"}, {\"body\":\"Buy Macbook pro\"}, {\"category_id\":2}, {\"id\":3,  
\"title\":null, \"body\":null, \"category_id\":null}, {\"id\":4,  
\"title\":\"test\"}, {\"body\":\"body\"}, {\"category_id\":2}]}"
```

10. In our show method, we can directly call our resource ID:

```
1.9.3p327 :011 > RestClient.get 'http://localhost:3000/api/v1/  
notes/1'  
=> "{\"note\":{\"id\":1,\"title\":\"First Note\",\"body\":\"Buy a  
new ram\"}, {\"category_id\":1}}"
```

Objective complete – mini debriefing

In this task, we modified our serializer to add the note and category association to it. As you will notice, the association here looks slightly different from our traditional model association:

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

```
class Category < ActiveRecord::Base
  has_many :notes
end
```

In the case of the serializer, the same association looks as follows:

```
class NoteSerializer < ActiveModel::Serializer
  embed :id

  attributes :id, :title, :body
  has_one :category
end
```

As opposed to models, serializers are not concerned with the ownership of a record and rather focus on multiplicity. This means if many notes have one category, the serializer still treats it as a multiple record with the value of 1. So, `belongs_to` makes way for `has_one` in serializers; it is just a different perspective to the same concept of association. The `embed :id` parameter will give access to the `category_id` field so that we do not have to worry about the attributes explicitly. The associated data is also embedded inside our JSON hash. Hence, the serializer will generate a nested JSON hash for an embedded association data. The following code snippet is the JSON object that is returned when we access `localhost:3000/api/v1/notes`:

```
{"notes": [{"id": 1, "title": "First Note", "body": "Buy a new ram", "category_id": 1}, {"id": 2, "title": "Second Note", "body": "Buy Macbook pro", "category_id": 2}]}
```

Embedding an association also gives the advantage of access to the entire category object from the note. This removes the need for another serializer for the categories. Also, in our use case, we have the `has_one` association. In case we want a `has_many` association, the `embed` will change as follows:

```
embed :ids
has_many :categories
```

This will supply an array of `category_id` fields to each record of notes. Now that we have formatted our data in the JSON format, we can secure our API.

Creating an OAuth2 provider

The most important reason underlying API development is the creation of the developer community. The applications contributed by different developers not only increase the popularity of the app, but also bring out several creative things people can do with the data; Twitter API is one such example. People have made amazing desktop clients and mobile apps that analyze tweets for trends and sentiments based on data. However, all these applications need to be genuine and should not spam the users. In order to avoid that, we will allow only OAuth-authorized applications to build clients for our API. Therefore, we will have to create an OAuth2 provider.

Prepare for lift off

Before we start working on this task, we will install devise. For the most part, the devise installation is pretty standard. In this case, we will use devise with warden as we will allow token-based authentication via warden using our doorkeeper gem:

Gemfile

```
gem 'devise'  
gem 'warden'
```

However, as Rails API removes the middleware layer and devise has some middleware dependencies, we will have to include them in our application controller:

```
app/controllers/application_controller.rb  
  
class ApplicationController < ActionController::API  
  include ActionController::MimeResponds  
  include ActionController::ImplicitRender  
end
```

ActionController::MimeResponds includes the respond_to and respond_with methods of Rails. ActionController::ImplicitRender includes methods such as default_render, method_for_action, and send_action. We also need to include the middleware flash module for our application to work. Doorkeeper uses Flash to display notices and alerts as shown in the following code:

```
config/application.rb  
  
module Notely  
  class Application < Rails::Application  
    config.middleware.use ActionDispatch::Flash  
  end  
end
```

Engage thrusters

We will make our application an OAuth2 provider in the following steps:

1. We will use the doorkeeper gem to create our OAuth2 provider:

```
Gemfile  
gem 'doorkeeper', '~> 0.7.0'  
  
notely$ bundle install
```

2. We will run the doorkeeper generator once the gem is bundled successfully:

```
$ rails generate doorkeeper:install  
      create config/initializers/doorkeeper.rb  
      create config/locales/doorkeeper.en.yml  
      route use_doorkeeper
```

3. This will create an initializer, a locale file, and add a route for endpoints in our application.
4. The doorkeeper gem also generates a migration. It creates a table to store OAuth access tokens and access grants:

```
notely$ rails generate doorkeeper:migration  
      create db/migrate/20131222100518_create_doorkeeper_tables.  
rb
```

5. Now, create tables with the rake task:

```
notely$ rake db:migrate  
== CreateDoorkeeperTables: migrating =====  
=====  
-- create_table(:oauth_applications)  
  -> 0.1671s  
-- add_index(:oauth_applications, :uid, {:unique=>true})  
  -> 0.2556s  
-- create_table(:oauth_access_grants)  
  -> 0.1107s  
-- add_index(:oauth_access_grants, :token, {:unique=>true})  
  -> 0.2004s  
-- create_table(:oauth_access_tokens)  
  -> 0.1109s  
-- add_index(:oauth_access_tokens, :token, {:unique=>true})  
  -> 0.2001s
```

```
-- add_index(:oauth_access_tokens, :resource_owner_id)
-> 0.2004s
-- add_index(:oauth_access_tokens, :refresh_token,
{:unique=>true})
-> 0.1899s
== CreateDoorkeeperTables: migrated (1.4364s) =====
=====
```

6. We will need to modify the initializer created here and make it use warden in order to access the user's resource from devise's `current_user` method:

`config/initializers/doorkeeper.rb`

```
Doorkeeper.configure do
  orm :active_record

  resource_owner_authenticator do
    current_user || warden.authenticate! (:scope => :user)
  end
end
```



Be sure to comment out or delete the following line from the code, else it will raise an error during execution:

```
#raise "Please configure doorkeeper resource_owner_
authenticator block located in #__FILE__"
```

7. We need to create a method to access the resource for doorkeeper to identify whether the logged-in user is authenticated against a valid application or not. This means it defines the owner of `access_token` which our application returns to each user:

```
app/controllers/application_controller.rb

class ApplicationController < ActionController::API
  include ActionController::MimeResponds
  include ActionController::ImplicitRender

  def current_resource_owner
    User.find(doorkeeper_token.resource_owner_id) if
      doorkeeper_token

  end
end
```

8. We can now protect our API methods using the `doorkeeper_for` method:

```
app/controllers/api/v1/notes_controller.rb
class NotesController < ApplicationController
  before_action :set_page, only: [:show, :edit, :update, :destroy]
  doorkeeper_for :index, :show, :update, :create
```

9. However, different methods require different levels of access. In order to abstract the different access levels, the `doorkeeper` gem has scopes. When a client requests for access, allowed actions are displayed. So first, we will have to enable the scopes in our `doorkeeper` initializer:

```
config/initializers/doorkeeper.rb

Doorkeeper.configure do

  orm :active_record

  resource_owner_authenticator do

    current_user || warden.authenticate! (:scope => :user)
  end

  default_scopes :public
  optional_scopes :write, :update
end
```

10. We have defined the following two scopes:

- ❑ **Public:** This scope is for all the data that is publicly available
- ❑ **Write and update:** This scope is only for users who are authenticated against the API

11. We need to add these scopes to the controller to bring them into play:

```
$app/views/api/v1/notes_controller.rb
class NotesController < ApplicationController
  before_action :set_page, only: [:show, :edit, :update, :destroy]
  doorkeeper_for :index, :show,      :scopes => [:public]
  doorkeeper_for :update, :create,   :scopes => [:write, :update, :destroy]
```

Objective complete – mini debriefing

In the preceding task, we first prepared our application with devise and added some middleware components for devise and doorkeeper to function properly. We loaded this in our `application_controller.rb` and `application.rb` files. The `doorkeeper` gem is a solution to make our application an OAuth2 provider. We first installed and generated an initializer for `doorkeeper`.

We defined the object-relational modeling for the application. It even supports different versions of Mongoid. In our case, we use `active_record`. Hence, we will define it as follows:

```
config/initializers/doorkeeper.rb
Doorkeeper.configure do
  orm :active_record
```

We then added `resource_owner_authenticator`, which is where we connect devise and doorkeeper. We made doorkeeper use the `current_user` method of devise and used warden to connect to the devise methods for authentication:

```
# This block will be called to check whether the resource owner is
authenticated or not.
resource_owner_authenticator do
  #raise "Please configure doorkeeper resource_owner_authenticator
block located in #{__FILE__}"
  current_user || warden.authenticate! (:scope => :user)
end
end
```

We added `current_resource_owner` to check for the owner of the doorkeeper resource:

```
def current_resource_owner
  User.find(doorkeeper_token.resource_owner_id) if
  doorkeeper_token
end
```

Under the hood, `doorkeeper_token` accesses the token generated upon a successful authentication request and returns it:

```
def doorkeeper_token
  methods = Doorkeeper.configuration.access_token_methods
  @token ||= OAuth::Token.authenticate request, *methods
end
```

We finally added scopes so that we can protect the resource and give limited access to the different types of users based on their roles and ownership. For public methods, such as `index` and `show`, we defined a scope called `public`. We can define this for users who want to just read without logging in. For users who want to create notes, we added scopes called `write` and `update`. In order to activate these scopes, we added them as a `filter` method, `doorkeeper_for` in our controller so that they are checked before the methods are executed.

Generating API keys

Doorkeeper is a complete solution for API authorization as well as app management using the OAuth2 protocol. In the previous task, we added `doorkeeper` and configured it to our needs. In this task, we will see how to generate API keys and do some final integration with devise for authentication. Only logged-in users can create applications. This is a use case for when we want to give freedom to several developers to create applications using our API.

Engage thrusters

In the following steps, we will add the devise layer above `doorkeeper` and generate API keys for the first time:

1. We need to generate the polymorphic association and addition to the application owner:

```
notely$ rails generate doorkeeper:application_owner
      create  db/migrate/20131228141233_add_owner_to_application.rb'
```

2. Run the migration:

```
notely$ rake db:migrate
==  AddOwnerToApplication: migrating =====
=====
-- add_column(:oauth_applications, :owner_id, :integer,
{:null=>true})
      -> 0.1840s
-- add_column(:oauth_applications, :owner_type, :string,
{:null=>true})
      -> 0.2002s
-- add_index(:oauth_applications, [:owner_id, :owner_type])
      -> 0.4107s
==  AddOwnerToApplication: migrated (0.7954s) =====
=====
```

3. The table now looks like the following:

```
create_table "oauth_applications", force: true do |t|
  t.string    "name",                      null: false
  t.string    "uid",                       null: false
  t.string    "secret",                     null: false
  t.string    "redirect_uri", limit: 2048, null: false
  t.datetime "created_at"
  t.datetime "updated_at"
  t.integer  "owner_id"
  t.string   "owner_type"
end
```

4. We will add `enable_application_owner` in order to enable the ownership of created applications. This is false by default because we would not want the application owner to confirm his/her membership to use the application. If the value is true, the owner will be asked to authenticate against the application just like other users:

```
Doorkeeper.configure do
  orm :active_record

  resource_owner_authenticator do
    current_user || warden.authenticate! (:scope => :user)
  end

  enable_application_owner :confirmation => false

  default_scopes  :public
  optional_scopes :write, :update
end
```

5. In order to access the `current_user` object, we need to authenticate and log in before we create the apps:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
  has_many :oauth_applications, class_name:
'Doorkeeper::Application', as: :owner
end
```

6. We will have to modify our controller so that we can access `current_user` in it. As `doorkeeper` is a Rails engine, we will create a folder called `OAuth` inside our controllers and copy the application controller to the folder:

```
notely$mkdir oauth
notely$cd oauth

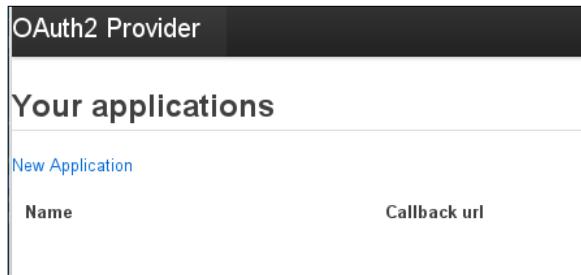
app/controllers/oauth/applications_controller.rb
class OAuth::ApplicationsController < Doorkeeper::ApplicationsController
  before_filter :authenticate_user!

  def index
    @applications = current_user.oauth_applications
  end

  # only needed if each application must have owner
  def create
    @application = Doorkeeper::Application.new(application_params)
    @application.owner = current_user if Doorkeeper.configuration.confirm_application_owner?
    if @application.save
      flash[:notice] = I18n.t(:notice, :scope => [:doorkeeper, :flash, :applications, :create])
      respond_with [:oauth, @application]
    else
      render :new
    end
  end

end
```

7. We will now boot our server and log in. In order to create the OAuth2 application, we will have to browse to `localhost:3000/oauth/applications`. We will be presented with the application management dashboard as shown in the following screenshot:



8. For example, in all OAuth2 providers, we will have to add the application name and callback URL for our application. It is better to enter either a real and valid domain name or a domain that resolves at localhost (lvh.me).

The screenshot shows a web interface for creating a new application. At the top, it says "OAuth2 Provider". Below that, "New application" is displayed. There are two input fields: "Name" containing "TweetNote" and "Redirect uri" containing "http://tweetnote.com". To the right of the redirect URI field is a note: "Use urn:ietf:wg:oauth:2.0:oob for local tests". On the far right, under "Actions", there is a link "Back to application list". At the bottom of the form are two buttons: "Submit" (in blue) and "Cancel".

9. Lastly, we will add a devise authentication so that a user needs to pass the username and password to get an access token:

```
config/initializers/doorkeepr.rb

resource_owner_from_credentials do |routes|

  request.params[:user] = { :email =>
    request.params[:username], :password =>
    request.params[:password] }

  request.env["devise.allow_params_authentication"] =
  true

  request.env["warden"].authenticate! (:scope => :user)

end
```

Objective complete – mini debriefing

Up until now, we have created a devise-based authentication and doorkeeper authorization for the applications. However, we had to allow the users to create authorizable applications. The doorkeeper project resides on GitHub (<https://github.com/applicake/doorkeeper>) and the documentation can be found at the project wiki (<https://github.com/applicake/doorkeeper/wiki>). There are several other tutorials that can be found on it, including ones to build a client application.

Doorkeeper allows us to create ownership for the applications that developers want to create. We ran a generator task in doorkeeper to create the migration for that:

```
notely$rails generate doorkeeper:application_owner
```

This generates the following migration by adding `owner_id` and `owner_type` to the `oauth_applications` table:

```
class AddOwnerToApplication < ActiveRecord::Migration
  def change
    add_column :oauth_applications, :owner_id, :integer, :null => true
    add_column :oauth_applications, :owner_type, :string, :null => true
    add_index :oauth_applications, [:owner_id, :owner_type]
  end
end
```

We then enabled application ownership. We have set `confirmation` to `false` so that the application owner does not need to connect and confirm the app before using it.

```
enable_application_owner :confirmation => false
```

If we change this to `true`, then even the application owner will have to grant access to the application in order to use it.

We then created an association between the user model and doorkeeper's `oauth_applications` model:

```
has_many :oauth_applications, class_name: 'Doorkeeper::Application',
as: :owner
```

After the association, we had to make sure a logged-in user creates the application, hence we added `before_filter`. We also used the `current_user` method of devise to call all the applications by a particular user in the index page:

```
before_filter :authenticate_user!

def index
  @applications = current_user.oauth_applications
end
```

Our `create` method also ensures that every application has to have an owner using the `confirm_application_owner?` method. This is a use case for when we want a lot of users to use our application:

```
def create
  @application = Doorkeeper::Application.new(application_params)
```

```
    @application.owner = current_user if Doorkeeper.configuration.  
confirm_application_owner?  
    if @application.save  
      flash[:notice] = I18n.t(:notice, :scope => [:doorkeeper,  
        :flash, :applications, :create])  
      respond_with [:oauth, @application]  
    else  
      render :new  
    end
```

Then we went ahead and created an application. An application ID and secret are created upon submitting the application form. The callback URL is generally a valid URL because the application has to return to it after authorization:

The screenshot shows a web interface for managing OAuth2 applications. At the top, a dark header bar reads "OAuth2 Provider". Below it, a white main area has a title "Application: TweetNote". On the left, there are several configuration fields with their values highlighted in orange boxes:

- Callback url:** `http://tweetnote.com`
- Application Id:** `5885177a47013eda93464fb764567d1ebe8ba411001e798fc1745d22e839ce36`
- Secret:** `8f131eceab62a2bdf149f74a33c0bd24ad616d84617ab10673c1a433ef81c7ba`
- Link to authorization code:** A blue "Authorize" button.

On the right side, there is a column titled "Actions" with three buttons: "List all", "Edit", and "Remove".

In order to test our API, we will use curl and send a request. This request includes `client_id`, `client_secret`, `username`, and `password` and is formatted as follows:

```
$curl -i http://localhost:3000/oauth/token \  
-F grant_type=password \  
-F client_id="5885177a47013eda93464fb764567d1ebe8ba411001e798fc1745d  
22e839ce36" \  
-F client_secret="8f131eceab62a2bdf149f74a33c0bd24ad616d84617ab10673  
c1a433ef81c7ba" \  
-F username="saurabh.a.bhatia@gmail.com" \  
-F password="safew123"
```

We can view the response in the following screenshot:

```
rwub@rwub:~/rails4-book/book/629405_Chapter_08/project-8$ curl -i http://localhost:3000/oauth/token -F grant_type=password
client_id=5885177a47013eda9346fb764567d1ebcb411001e798fc1745d2e839ce36"      -F client_secret="8f131eceab62a2bdf149f74a33c0bd2
84617ab10673c1a433ef81c7ba"      -F username="saurabh.a.bhatia@gmail.com"      -F password="safeW123"
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Cache-Control: no-store
Pragma: no-cache
ETag: "ba8fd2350d29226243f3d1c03bb6a20a"
X-Request-Id: 92b4f3a7-cd2e-423a-adc3-306565253237
X-Runtime: 0.579782
Transfer-Encoding: chunked

{"access_token":"7dcc33d2a9616d1b34085862aacf0676efc3f3ea4cb2d32f63354ddf328273c7","token_type":"bearer","expires_in":7200,"scope":
```

The response of the API includes `access_token`, `token_type`, and `expires_in` (expiry time). This means our application is successfully authenticating as well as authorizing using OAuth:

```
{
  "access_token": "54ca3950883abcb50a4e1e04dff94114dc3e561b452eaed753957
  9e3c3f12026",
  "token_type": "bearer",
  "expires_in": 7200,
  "scope": "public"
}
```

This check is done using the rule that we added in the previous step:

```
resource_owner_from_credentials do |routes|
```

The previous (`resource_owner_from_credentials`) method matches the supplied credentials using the devise user model.

Securing the application

Security is one of the primary concerns of an API application. We have already provided some level of security with authentication and authorization. However, we still need to add extra layers of security to our application. Doorkeeper and warden allow token-based authentication, and hence a user has to have an authentication token. Also, the application is authenticated against the application ID and secret.

Engage thrusters

We will now add some security-related tricks to our application by performing the following steps:

1. The first level of security we will provide is against session fixation. In our devise initializer, we need to add the following lines:

```
config/initializers/devise.rb
```

```
Warden::Manager.after_authentication do |record, warden, options|
  warden.request.session.try(:delete, :_csrf_token)
end
```

2. We will now set up the session timeout in our application so that the session is deleted after the specified time interval:

```
app/models/user.rb
```

```
class User < ActiveRecord::Base
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable,
         :validatable, :timeoutable, :timeout_in =>
  15.minutes

  has_many :oauth_applications, class_name:
    'Doorkeeper::Application', as: :owner
end
```

3. SQL injection attacks are pretty common in web applications. However, Rails provides enough protection against SQL injection. Rails already provides one level of protection in the controllers by whitelisting parameters:

```
app/views/api/v1/notes_controller.rb
```

```
def note_params
  params.require(:note).permit(:title,
    :body, :category_id) if params[:note]
end
```

4. In case we need to pass a parameter, we need to pass it as a string:

```
Note.where(:title => "'#{params[:title]}'"')
```

5. The right way to avoid SQL injection using the parameter is as follows:

```
Note.where("title=?", title)
```

Objective complete – mini debriefing

Security is an extremely critical aspect of our applications in today's world. In this task, we looked at some of the ways Rails already provides security to the application by default, and some other ways in which we can secure our application.

The first thing we looked at was session fixation. Wikipedia defines session fixation as follows:

In computer network security, session fixation attacks are an attempt to exploit the vulnerability of a system, which allows one person to fixate another person's session identifier. Most session fixation attacks are web based, and most rely on session identifiers being accepted from URLs or POST data.

Devise out of the box is quite secure. However, this scenario can occur in the following two cases:

- ▶ When the attacker uses subdomain cookies to enter the target session
- ▶ When the attacker exploits the same Wi-Fi network for fixation

In order to avoid this, we delete the following unique session CSRF token as soon as the authentication is complete:

```
warden.request.session.try(:delete, :_csrf_token)
```

The preceding line of code will clear the CSRF token. So, if an attacker is trying to steal the token, they are not able to, and hence the session is secure.

Another way that we looked at was timing out our sessions. Sessions are most susceptible to attack when they have some idle time on them. In order to avoid these attacks, we can clear the session. The `timeoutable` module in devise allows us to define when to expire the session:

```
:timeoutable, :timeout_in => 15.minutes
```

We defined in our application that the session should expire if it is idle for 15 minutes. This setting should use a much higher value in a real-world application because we would not want a user to log in again if they are idle for 15 minutes.

Lastly, we secured our application against SQL injection. Rails has been vulnerable to SQL injections owing to the mass assignment parameters in 3.2.x Versions, and there were serious security concerns related to it. Rails 4 sanitizes the parameters out of the box by using the standard blacklisting and whitelisting technique. Only the whitelisted parameters are allowed to pass to the controller. However, the nature of injection attacks is such that people can still insert malicious SQL statements inside params and allow them to be passed inside the query string.

Wikipedia defines SQL injection as follows:

SQL injection is a code injection technique used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution.

In Rails, a lot of times parameters are directly passed from a query string to the active record query interface using the `params[:title]` format. This makes the SQL statement vulnerable as someone can pass a string with an SQL statement, such as OR or AND, and execute SQL inside it.

We rephrased this query to a different format. We will first pass the SQL statement into a variable. Then, instead of directly passing the query string, we will sanitize the variable and pass it to a query as follows:

```
Note.where("title=?", title)
```

Another way is to pass the variable through the `sanitize_sql()` method before passing it to the query.

Also, in a real-world application, it is highly advisable to use an SSL certificate in order to provide secure access to your server, especially for transactions that involve passing your application's `application_id` and `application_secret`.

Mission accomplished

We have successfully created an API only application with an OAuth2 provider. We also looked at various security aspects in this project.

Some of the areas we covered in this project are as follows:

- ▶ We used the Rails API gem to create an API only application that does not contain rack middleware modules and frontend modules.
- ▶ We adapted some of the methods to Rails 4, for example, strong parameters.
- ▶ We used a JSON serializer to create clean JSON APIs that are interactive. We can read, write, and update the data using JSON.
- ▶ We also looked at how associations work in the serializer and how it is different from the regular models.
- ▶ We also saw how to read and post to the API using a REST client.
- ▶ Once we had a fully functional API in place, we used the `doorkeeper` gem in order to protect these API methods.

- ▶ We saw how doorkeeper in conjunction with devise turned the application into a full-fledged OAuth2 provider, thus providing it with an authentication as well as a conditional authorization framework.
- ▶ We created our own application using the newly created OAuth framework.
- ▶ We also worked on how to secure our application from different types of attacks.

Hotshot challenges

We have our API in place and a lot of functionalities to play around with. However, we would still like to take things to the next level with the following exercises:

- ▶ Allow and delete notes in the API and delete a note using the REST client.
- ▶ A different mime type to the API post method. By this I mean that you should allow a user to post an image using the post method in our API.
- ▶ Make sure that this method works only with Ajax requests using doorkeeper.
- ▶ Create a client application using HTTParty to read the notes of a user.
- ▶ Log in and authorize using the devise token authentication and the OAuth2 provider.

Project 9

Video Streaming Website using Rails and HTML5

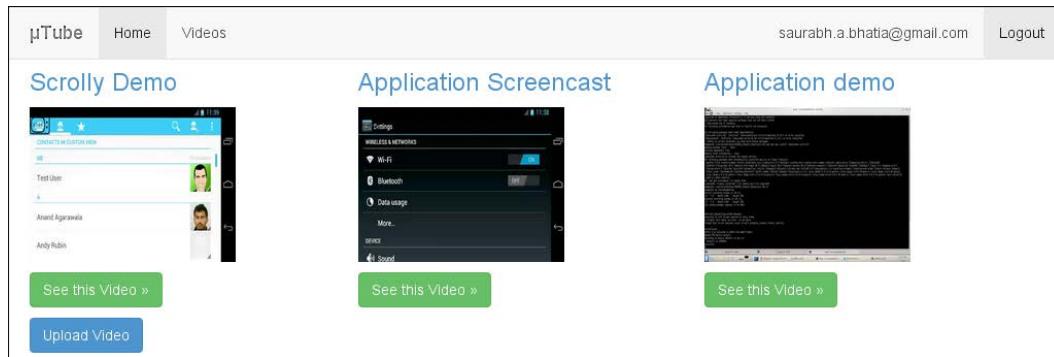
Video as a medium is quite appealing to a lot of users. It is a very effective way of communication, and the effect can be very long lasting. YouTube (<http://www.youtube.com>), Vimeo (<http://www.vimeo.com>), Dailymotion (<http://www.dailymotion.com>), and Khan Academy (<http://www.khanacademy.org>) are some of the most popular sites where a variety of content exists. Advertising, raising awareness, organizing campaigns, distributing films, and providing education are some of the most common uses of these. This has increased the accessibility of the content and allowed content creators of various languages to reach a very wide and diverse set of audiences.

Mission briefing

This project is a video-streaming website where a user uploads the video and the video is encoded to a HTML5 friendly format. We will also take screenshots of the video post their upload so that we can make thumbnails out of it. We will work on caching and performance improvement with videos. We will also take a look at process queues in Rails.

Why is it awesome?

A lot of ideas have been tried around video. With the advent of HTML5, the video standards are becoming much more flexible and device friendly. HTML5 reduces external dependencies and plugins in order to display and run videos. This will make the video and audio protocols more standardized and open. We will use some standards that work seamlessly with HTML5 video and make sure it works on different devices. Once this is in place, we will cache the video and text. We will use **Russian Doll caching**, a technique introduced in Rails 3.2 but carried forward in Rails 4. We will also see queues in Rails. We will allow our application to simultaneously process videos as jobs. The final project screen with a list of videos will appear as shown in the following screenshot:



At the end of this project, we will have a basic video-streaming web application.

Your Hotshot objectives

While building this application, we will have to go through the following tasks:

- ▶ Uploading the video
- ▶ Encoding the video
- ▶ Displaying the video panel and playing the video
- ▶ Caching the content – text and video
- ▶ Queuing the job

Mission checklist

We need the following installed on the system and also need to sign up for the API keys before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0+

- ▶ MySQL
- ▶ FFmpeg
- ▶ Devise
- ▶ Git
- ▶ Redis
- ▶ Sidekiq
- ▶ jQuery
- ▶ Video.js
- ▶ Bootstrap 3.0

Uploading the video

We will begin our project with video-uploading methods. We have already seen file uploading with the `carrierwave` gem in our previous projects (*Project 3, Creating an Online Social Pinboard*). In this project, we will take it one step forward by uploading videos.

We will also add the `friendly_id` gem to our application in order to create slugs:

```
Gemfile
gem 'carrierwave', :github => "jnicklas/carrierwave"
gem 'friendly_id', '5.0.3'
gem 'anjlab-bootstrap-rails', :require => 'bootstrap-rails',
                               :github => 'anjlab/bootstrap-rails',
                               :branch => '3.0.0'
```

Only Version 5.0.3 `friendly_id` is compatible with Rails 4.1. Also, at this step, make sure you have `devise` installed and have generated a user model to handle user authentication.

Engage thrusters

We will start by installing Rails API and generating our skeleton application:

1. We will first generate a video model and controller. Be sure to write tests before that, as follows:

```
mutube$ rails g scaffold video title:string description:string
      invoke  active_record
      create    db/migrate/20140105125840_create_videos.rb
      create    app/models/video.rb
      invoke  test_unit
```

```
create      test/unit/video_test.rb
create      test/fixtures/videos.yml
invoke    resource_route
  route      resources :videos
invoke    scaffold_controller
create      app/controllers/videos_controller.rb
invoke    erb
create      app/views/videos
create      app/views/videos/index.html.erb
create      app/views/videos/edit.html.erb
create      app/views/videos/show.html.erb
create      app/views/videos/new.html.erb
create      app/views/videos/_form.html.erb
invoke    test_unit
create      test/functional/videos_controller_test.rb
invoke    helper
create      app/helpers/videos_helper.rb
invoke    test_unit
create      test/unit/helpers/videos_helper_test.rb
invoke    assets
invoke    coffee
create      app/assets/javascripts/videos.js.coffee
invoke    scss
create      app/assets/stylesheets/videos.css.scss
invoke    scss
identical   app/assets/stylesheets/scaffolds.css.scss
```

2. Once we have the skeleton for the video, we will add the carrierwave gem to Gemfile and run bundle install:

```
Gemfile
gem 'carrierwave', :github => "jnicklas/carrierwave"
```

3. We will then generate the video uploader using the carrierwave generator:

```
mutube$rails g uploader Video
create  app/uploaders/video_uploader.rb
```

4. Mount the video uploader on the video model:

```
app/model/video.rb
class Video < ActiveRecord::Base
  mount_uploader :video, VideoUploader
  extend FriendlyId
  friendly_id :title, use: :slugged
end
```

5. Now we will add a column for video file parameters to our videos table:

```
class AddVideoToVideos < ActiveRecord::Migration
  def change
    add_column :videos, :media, :string
  end
end
```

6. We also need to pass the parameters for the video as a whitelist in our controller and add the friendly_id association to our set_video action:

```
app/controllers/videos_controller.rb
private
  # Use callbacks to share common setup or constraints between
  # actions.
  def set_video
    @video = Video.friendly.find(params[:id])
  end
  # Never trust parameters from the scary internet, only allow
  # the white list through.
  def video_params
    params.require(:video).permit(:title, :description, :media,
      :media_cache)
  end
```

7. We will edit the form to add the upload field for the video:

```
app/views/_form.html.erb
<%= form_for(@video, :html => {:multipart => true}) do |f| %>
  <% if @video.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@video.errors.count, "error") %>
      prohibited this video from being saved:</h2>
      <ul>
        <% @video.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
```

```
<% end %>
<div class="form-group">
  <label>Title</label>
  <%= f.text_field :title, :class => "form-control", :placeholder => "title" %>
</div>
<div class="form-group">
  <label>Description</label>
  <%= f.text_area :description, :class => "form-control" %>
</div>
<div class="form-group">
  <label for="InputFile">Upload Video</label>
  <%= f.file_field :media %>
  <%= f.hidden_field :media_cache %>
  <p class="help-block"></p>
</div>
<%= f.submit "Save", :class => "btn btn-default" %> <%= link_to 'Cancel', videos_path, :class => "btn btn-danger" %>
<% end %>
```

8. Lastly, we will restrict our video formats to MP4, OGV, and AVI. This will only allow the whitelisted file formats to be uploaded:

```
app/uploaders/video_uploader.rb
# encoding: utf-8
class VideoUploader < CarrierWave::Uploader::Base
  include CarrierWave::MimeTypes
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end
  def extension_white_list
    %w(mp4 ogv avi)
  end
end
```

Objective complete – mini debriefing

This task was a recap of things we have already done in our past projects. We created model and controllers views for the video. We added the carrierwave uploader and restricted the formats for a video upload in order to avoid malicious uploads as shown in the following code:

```
def extension_white_list
  %w(mp4 ogv avi)
end
```

Please keep in mind that we have chosen the storage mechanism as **file only** for the sake of convenience. Video files are generally big in size, hence, we do not want to select them again and again. Therefore, we added `media_cache` to the `video_params` so that the form retains the selected video, even if the validation fails and the form reloads afterwards. There are several other mechanisms such as Amazon S3 and Rackspace Block Storage to store the files.

The form for video upload with the upload video file field looks like the following screenshot:

The screenshot shows a web form titled "Add a New video". The form has fields for "Title" (with placeholder "title") and "Description" (a large text area). Below these is a "Upload Video" section with a "Choose File" button and a message "No file chosen". At the bottom are "Save" and "Cancel" buttons. The top right corner of the form has a three-line menu icon.

Encoding the video

Video encoding should be part of the upload process. We need to encode the uploaded video files to an HTML5-friendly format, basically the MP4 format, which is fully implemented in the new HTML standard. We will use the `ffmpeg` on the system side and `carrierwave-video` extensions on our application side to do so. During the implementation of this process, we will also update the library for `carrierwave-video` to ensure it suits our needs.

Prepare for lift off

We will first install the dependencies for `ffmpeg`. We'll also need to install the Theora and Vorbis protocols for audio and video respectively.

1. The best way to install `ffmpeg` on Mac OS X is through the use of homebrew:

```
$ brew install ffmpeg --with-fdk-aac --with-ffplay --with-freetype  
--with-frei0r --with-libass --with-libvo-aacenc --with-libvorbis  
--with-libvpx --with-opencore-amr --with-openjpeg --with-opus  
--with-rtmpdump --with-schroedinger --with-speex --with-theora -  
with-tools
```

```
==> Installing dependencies for ffmpeg: texi2html, yasm, x264,
faac, lame, xvid, libpng, freetype, libogg, xz, libvorbis, theora,
libvpx, rtmp
==> Installing ffmpeg dependency: texi2html
```

2. Following are the instructions for building ffmpeg on Ubuntu:

```
mutube$ sudo apt-get -y install autoconf automake build-essential
git libass-dev libgpac-dev \
libsdl1.2-dev libtheora-dev libtool libva-dev libvdpau-dev
libvorbis-dev libx11-dev \
libxext-dev libxfixes-dev pkg-config texi2html zlib1g-dev
```

3. It is most preferable to install all dependencies by compiling them from the source. The first dependency is yasm, an assembler used by video and audio encoders:

```
$wget http://www.tortall.net/projects/yasm/releases/yasm-
1.2.0.tar.gz
yasm$ tar xvzf yasm.tar.gz
yasm$ sed -i 's#) ytasm.*#)##' Makefile.in &&
./configure --prefix=/usr &&
make
yasm$ make install
```

4. Check whether yasm is installed or not:

```
yasm $ sudo which yasm
/usr/bin/yasm
```

5. Now, we will install x264, the video encoder:

```
$wgetftp://ftp.videolan.org/pub/x264/snapshots/last_x264.tar.bz2
$tar xvjf last_x264.tar.bz2
x264$ ./configure --prefix="$HOME/ffmpeg_build" --bindir="$HOME/
bin" --enable-static
x264$ make
x264$ sudo make install
```

6. After the installation of the video encoder, we will install the audio encoder, aac. The newer version of ffmpeg uses aac instead of the earlier library, libfaac:

```
$ git clone git@github.com:mstorsjo/fdk-aac.git
$cd fdk-aac
fdk-aac$autoreconf -fiv
fdk-aac$ ./configure --prefix="$HOME/ffmpeg_build" --disable-
shared
fdk-aac$ make
fdk-aac$ make install
```

7. Next, we will add support for .mp3 audio:

```
$ sudo apt-get install libmp3lame-dev
```

8. We also need to add Opus's encoder and decoder support:

```
$ sudo apt-get install libopus-dev
```

9. We need support for V8/V9 video formats, so we will compile the libvpx project extracted from Android:

```
$ git clone http://git.chromium.org/webm/libvpx.git  
cd libvpx  
/configure --prefix="$HOME/ffmpeg_build" --disable-examples  
make  
make install
```

10. After all the dependencies are installed, we will compile ffmpeg using different protocol supports such as aac, x264, and x11 compatibilities:

```
$ git clone --depth 1 git://source.ffmpeg.org/ffmpeg  
$ cd ffmpeg  
ffmpeg$ PKG_CONFIG_PATH="$HOME/ffmpeg_build/lib/pkgconfig"  
ffmpeg$ export PKG_CONFIG_PATH  
ffmpeg$ ./configure --prefix="$HOME/ffmpeg_build" \  
--extra-cflags="-I$HOME/ffmpeg_build/include" --extra-ldflags="-  
L$HOME/ffmpeg_build/lib" \  
--bindir="$HOME/bin" --extra-libs="-ldl" --enable-gpl --enable-  
libass --enable-libfdk-aac \  
--enable-libmp3lame --enable-libopus --enable-libtheora  
--enable-libvorbis --enable-libvpx \  
--enable-libx264 --enable-nonfree --enable-x11grab  
ffmpeg$ make  
ffmpeg$ make install
```

11. We will test our ffmpeg installation with a simple command to check if aac support is installed or not:

```
$ ffmpeg -formats 2>&1 | grep aac  
configuration: --prefix=/home/rwub/ffmpeg_build --extra-  
cflags=-I/home/rwub/ffmpeg_build/include --extra-ldflags=-L/  
home/rwub/ffmpeg_build/lib --bindir=/home/rwub/bin --extra-libs=--  
ldl --enable-gpl --enable-libass --enable-libfdk-aac --enable-  
libmp3lame --enable-libopus --enable-libtheora --enable-libvorbis  
--enable-libvpx --enable-libx264 --enable-nonfree --enable-x11grab  
--enable-libfaac  
D aac raw ADTS AAC (Advanced Audio Coding)
```

12. The command will return the configuration details of AAC and hence we know that `ffmpeg` is installed properly.
13. For installation on Windows, the builds are available at <http://ffmpeg.zeranoe.com/builds/>. For using the archive, we need 7-zip installed on our machine. In order to install it, we need to download and unzip the archive first. From the `bin` folder inside our unzipped archive, we will see a file called `ffmpeg.exe`. We need to copy it to the path `C:/Tools/bin` in our filesystem.

Engage thrusters

In this task, we will encode our video during our upload process:

1. We will use a plugin called `carrierwave-video` that in turn uses the `streamio-ffmpeg` gem to connect to and subsequently use `ffmpeg` features. Before we proceed with its installation, we will customize it a bit. I have forked the original gem to my GitHub ID and cloned the repository on my local machine as follows:
`$ git clone https://github.com/saurabhbhattacharya/carrierwave-video.git`
2. We will change the `custom` option under default options, and remove `qscale` from the options:

```
carrierwave_video/lib/carrierwave/video/ffmpeg_options.rb
h[:custom] = '-qscale 0 -preset slow -g 30'
+           h[:custom] = "-strict experimental -preset slow -g
30"
```

3. We will also remove the default audio codec from MP4 and let `ffmpeg` autodetect the audio codec by itself:

```
carrierwave_video/lib/carrierwave/video/ffmpeg_options.rb
-           h[:audio_codec] = 'aac'
```

4. So our method now looks like the following:

```
lib/carrierwave/video/ffmpeg_options.rb
private
  def defaults
    @defaults ||= { resolution: '640x360', watermark: {} }.tap do
|h|
    case format
    when 'mp4'
      h[:video_codec] = 'libx264'
      h[:custom] = "-strict experimental -preset slow -g 30"
    when 'ogv'
      h[:video_codec] = 'libtheora'
      h[:audio_codec] = 'libvorbis'
      h[:custom] = '-b 1500k -ab 160000 -g 30'
    when 'webm'
      h[:video_codec] = 'libvpx'
```

```
    h[:audio_codec] = 'libvorbis'
    h[:custom] = '-b 1500k -ab 160000 -f webm -g 30'
  end
end
end
```

5. Once it is ready, we can commit and push these changes to the repository. We will now bundle directly from our forked repository to pick up the changes we just did:

```
Gemfile
gem "streamio-ffmpeg"
gem 'carrierwave-video', :github => 'saurabhbhatisa/carrierwave-
video'
```

6. We will add an encode process to encode our video to MP4:

```
app/uploaders/video_uploader.rb
def encode
  process encode_video: [:mp4, callbacks: { after_transcode: :set_
success } ]
end
```

7. In our `video_uploader.rb` file, we will have to include the CarrierWave video module. We will also have to add our `encode` method as a process to generate the MP4 version of the video:

```
app/uploaders/video_uploader.rb
class VideoUploader < CarrierWave::Uploader::Base
  include CarrierWave::MimeTypes
  include CarrierWave::Video
  storage :file

  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.
id}"
  end

  version :mp4 do
    process :encode
  end

  def encode
    process encode_video: [:mp4, callbacks: { after_transcode:
:set_success } ]
  end

  def extension_white_list
    %w(mp4 ogv avi)
  end
end
```

This will encode the video and convert it into MP4 once it is uploaded.

8. We would also like to add a watermark to our video after it is uploaded. This is to avoid plagiarism as much as possible. We will first set the path for the watermark image:

```
app/uploaders/video_uploader.rb
DEFAULTS = {
  watermark: {
    path: Rails.root.join('mutube.png')
  }
}
```

9. Now we will modify the `encode` method we previously wrote to add a watermark to the video:

```
app/uploaders/video_uploader.rb
def encode
  encode_video(:mp4, DEFAULTS) do |movie, params|
    if movie.height < 720
      params[:watermark][:path] = Rails.root.join('mutube.png')
    end
  end
end
```

10. So finally, our uploader looks like the following:

```
app/uploaders/video_uploader.rb
# encoding: utf-8
class VideoUploader < CarrierWave::Uploader::Base
  include CarrierWave::MimeTypes
  include CarrierWave::Video
  storage :file

  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  DEFAULTS = {
    watermark: {
      path: Rails.root.join('mutube.png')
    }
  }
  version :mp4 do
    process :encode
  end

  ]
  def encode
    encode_video(:mp4, DEFAULTS) do |movie, params|
      if movie.height < 720
        params[:watermark][:path] = Rails.root.join('mutube.png')
      end
    end
  end
end
```

```
    end
  end
end

def extension_white_list
  %w(mp4 ogv avi)
end
end
```

11. We will generate a screenshot for our video now. We will directly access and use the `streamio-ffmpeg` library in order to generate a screenshot and save it to the specified path:

```
app/models/video.rb
def video_screenshot
  screenshot_path = Rails.root+"/app/assets/images/
screenshots/#{self.slug}_#{self.id}.jpg"
  if FileTest.exists?(screenshot_path)
    @screenshot = screenshot_path
  else
    video_file = FFMPEG::Movie.new("#{Rails.root}/public"+self.
video.url(:mp4))
    @screenshot = video_file.screenshot("#{screenshot_path}")
  end
end
```

12. When we upload the video, we can see the following parameters:

```
Started POST "/videos" for 127.0.0.1 at 2014-04-02 20:33:45 +0800
Processing by VideosController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"PbLlrSy8ezaXELdP8VgLodxTeSy030pSaIhMiTbmoQY=",
"video"=>"another application video", "description"=>"another application video",
"video"=>#<ActionDispatch::Http::UploadedFile:0x0000001132a30 @tempfile=>#<Tempfile:/tmp/RackMu
ltpart20140402-3983-vwwmez>, @original_filename="out.ogv", @content_type="video/ogg", @headers
="Content-Disposition: form-data; name=\"video[video]\"; filename=\"out.ogv\"\\r\\nContent-Type:
video/ogg\\r\\n", "video_cache"=>"", "user_id"=>"1", "commit"=>"Save"}
  User Load (0.6ms)  SELECT `users`.* FROM `users` WHERE `users`.`id` = 1 ORDER BY `users`.`
id` ASC LIMIT 1
I, [2014-04-02T20:33:45.396194 #3983]  INFO -- : Running transcoding...
ffmpeg -y -i /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396442
025-3983-0041/mp4_out.ogv -vcodec libx264 -s 640x364 -strict experimental -preset slow -g 30 -
vf "movie=/home/rwub/rails4-book/book/62940S_Chapter_09/project-9/mutube.png [logo]; [in][logo]
overlay= [out]" -aspect 1.7582417582417582 /home/rwub/rails4-book/book/62940S_Chapter_09/proje
ct-9/public/uploads/tmp/1396442025-3983-0041/tmpfile.mp4

I, [2014-04-02T20:33:48.005392 #3983]  INFO -- : Transcoding of /home/rwub/rails4-book/book/629
40S_Chapter_09/project-9/public/uploads/tmp/1396442025-3983-0041/mp4_out.ogv to /home/rwub/rail
s4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396442025-3983-0041/tmpfile.mp4 su
ccceeded

  (0.3ms)  BEGIN
  Video Exists (0.5ms)  SELECT 1 AS one FROM `videos` WHERE `videos`.`slug` = 'another-applic
ation-video' LIMIT 1
  SQL (0.6ms)  INSERT INTO `videos` (`created_at`, `description`, `slug`, `title`, `updated_at`
, `user_id`, `video`) VALUES ('2014-04-02 12:33:48', 'another application video', 'another-appl
ication-video', 'another application video', '2014-04-02 12:33:48', 1, 'out.ogv')
  User Load (0.8ms)  SELECT `users`.* FROM `users` WHERE `users`.`id` = 1 ORDER BY `users`.`
```

Objective complete – mini debriefing

In this task, we began with the installation of `ffmpeg`. Their website (<http://ffmpeg.org>) defines it as:

FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video.

We installed all the dependencies required to run `ffmpeg` and compiled it from the source. Be sure to check more command-line options at the following links:

- ▶ Generic options: <http://ffmpeg.org/ffmpeg.html#Generic-options>
- ▶ Customizing videos: <http://ffmpeg.org/ffmpeg.html#Video-Options>
- ▶ Advanced options: <http://ffmpeg.org/ffmpeg.html#Advanced-Video-Options>

We then customized the `carrierwave-video` (<https://github.com/rheaton/carrierwave-video>) plugin. We updated the protocols for video and audio encoding to `aac`. The earlier version of `ffmpeg` used a `libfaac` to connect to the audio protocol. It has been deprecated in the newer version. We allowed `ffmpeg` to autodetect the audio protocol and encode accordingly. Next, we added the ability to watermark our videos in our application. The uploader will call the watermark image and send it along with the encoding command. You can see the following command, which is being fired to encode our video to MP4:

```
ffmpeg -y -i /home/user/mutube/public/uploads/tmp/1389051112-8075-3957/
mp4_scroll_index.mp4 -vcodec libx264 -s 640x358 -strict experimental
-preset slow -g 30 -vf "movie=/home/user/mutube/mutube.png [logo]; [in]
[logo] overlay= [out]" -aspect 1.7877094972067038 /home/user/mutube/
public/uploads/tmp/1389051112-8075-3957/tmpfile.mp4
```

We also generated a screenshot. We first checked whether the file already exists or not. We used `FileTest` and pointed it to the exact path:

```
screenshot_path = "#{Rails.root}/app/assets/images/
screenshots/"+"#{self.slug}_#{self.id}.jpg"

if FileTest.exists?(screenshot_path)
  @screenshot = screenshot_path
```

To generate the screenshot, we used the `streamio-ffmpeg` library:

```
video_file = FFMPEG::Movie.new("#{Rails.root}/public/" + self.video.
url(:mp4))

@screenshot = video_file.screenshot("#{Rails.root}/app/assets/
images/screenshots/"+"#{self.slug}_#{self.id}.jpg")
end
```

Displaying the video panel and playing the video

Displaying and playing videos has been a challenge for a long time. Flash has dominated the game all along and still powers most of the major websites. However, flash has not been good at optimization for mobile devices. HTML5 then released a fresh set of standards including MP4 and OGV for video, AAC, and OGG for audio. We will use these standards to display the video along with the `video.js` library.

Engage thrusters

We will display the uploaded videos in this task:

1. We will first download the latest version of `video.js` (download it from <http://www.videojs.com/downloads/video-js-4.4.2.zip>) and unzip it. We will place it under our `javascripts` folder under `app/assets/`. We will load it to our manifest file:

```
app/assets/javascripts/application.js
//= require jquery
//= require jquery_ujs
//= require twitter/bootstrap
//= require video
//= require turbolinks
//= require_tree
```

2. We will copy the `video-js.css` file to the `stylesheets` folder under `app/assets/` and add its reference to the CSS manifest file:

```
app/assets/stylesheets/application.css
*= require_self
*= require twitter/bootstrap
*= require video-js
*= require sticky-footer-navbar
*= require font-awesome
*= require_tree .
*/
```

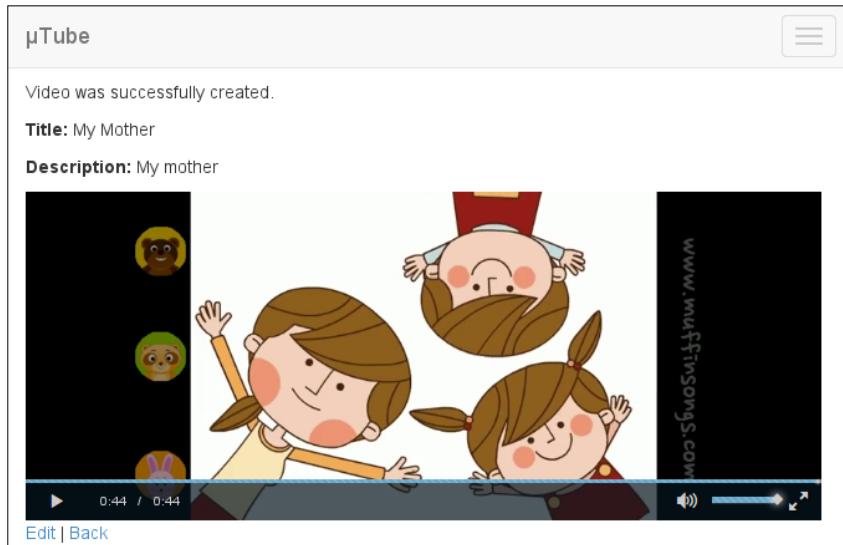
3. We will also have to place `video-js.swf` in our Rails application root.
4. We will start by adding the initialization code to the `show.html.erb` file:

```
app/views/videos/show.html.erb
<script>
  videojs.options.flash.swf = "#{Rails.root}/video-js.swf";
</script>
```

5. We will need to create a video element in our `show.html.erb` and load `video.js` default skin. We will also load the MP4 version of the video:

```
app/views/videos/_show.html.erb
<div class="row">
  <div class="col-lg-8">
    <h3><%= @video.title %></h3>
    <video id="video_1" class="video-js vjs-default-skin"
controls preload="none" width="640" height="264"
data-setup="{}">
      <source src="<%= @video.video.url(:mp4) %>" type='video/mp4' />
    </video>
    <br/>
    <p><%= @video.description %></p>
  </div>
</div>
```

We will now reload our page to see the video:



6. In order to increase the engagement of our website, we will display all the videos other than the current video in our show page.
7. We will first write a class method to find all videos other than the current video being viewed:

```
app/models/video.rb
def self.get_other_videos(video_id)
  videos = Video.where.not(id: video_id) rescue []
  return videos
end
```

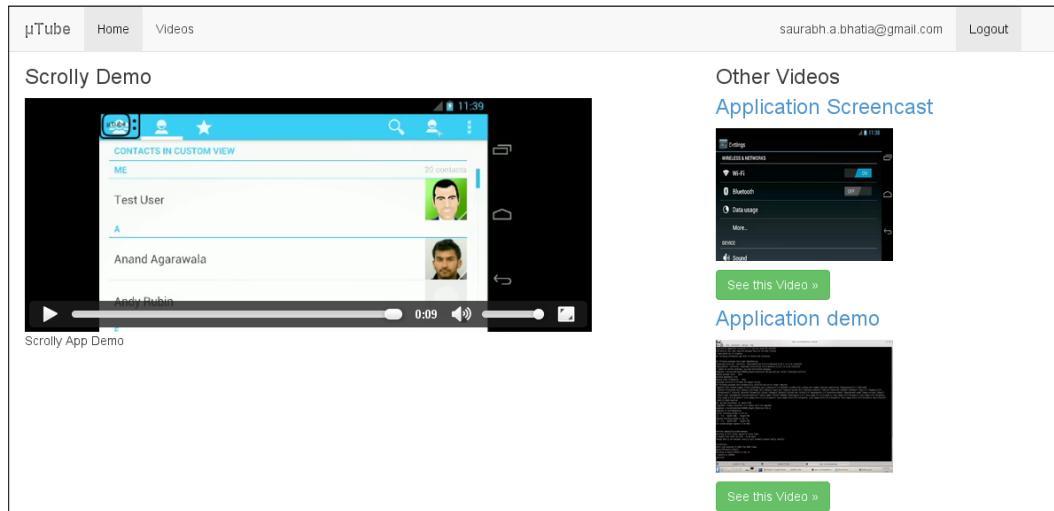
8. We will make a call on this method in the `videos_controller.rb` file:

```
app/controllers/videos_controller.rb
def show
  @videos = Video.get_other_videos(@video.id)
end
```

9. Then, we will loop through these videos and display them as a list on the right-hand side of the page. We will also create a helper to display the screenshot for each video:

```
app/helpers/videos_helper.rb
module VideosHelper
  def display_screenshot(video_slug, video_id)
    "screenshots/#{video_slug}_#{video_id}.jpg"
  end
end
app/views/app/show.html.erb
<div class="row">
  <div class="col-lg-8">
    <h3><%= @video.title %></h3>
    <video id="example_video_1" class="video-js vjs-default-skin" controls preload="none" width="640" height="264" data-setup="{}">
      <source src="<%= @video.video.url(:mp4) %>" type='video/mp4' />
    </video>
    <br/>
    <p><%= @video.description %></p>
  </div>
  <div class="col-lg-4">
    <h3>Other Videos</h3>
    <% @videos.each do |video| %>
      <h3><%= link_to video.title, video %></h3>
      <% video.video_thumbnail %>
      <p><%= image_tag display_screenshot(video.slug, video.id) , :width => 200, :height => 150 %></p>
      <p><%= link_to 'See this Video &raquo;', video, :class=>"btn btn-success" %></p>
    <% end %>
  </div>
</div>
```

10. The following screenshot displays the videos on the right-hand side of the screen:



Objective complete – mini debriefing

In this task, we used the `video.js` library to display the uploaded and encoded video. We added the appropriate JavaScript and CSS to the `application.js` and `application.css` files. After that, we initiated `videojs` with options. We kept `video-js.swf` in order to keep a fallback for browsers that do not support the HTML5 video as yet:

```
<script>
  videojs.options.flash.swf = "#{Rails.root}/video-js.swf";
</script>
```

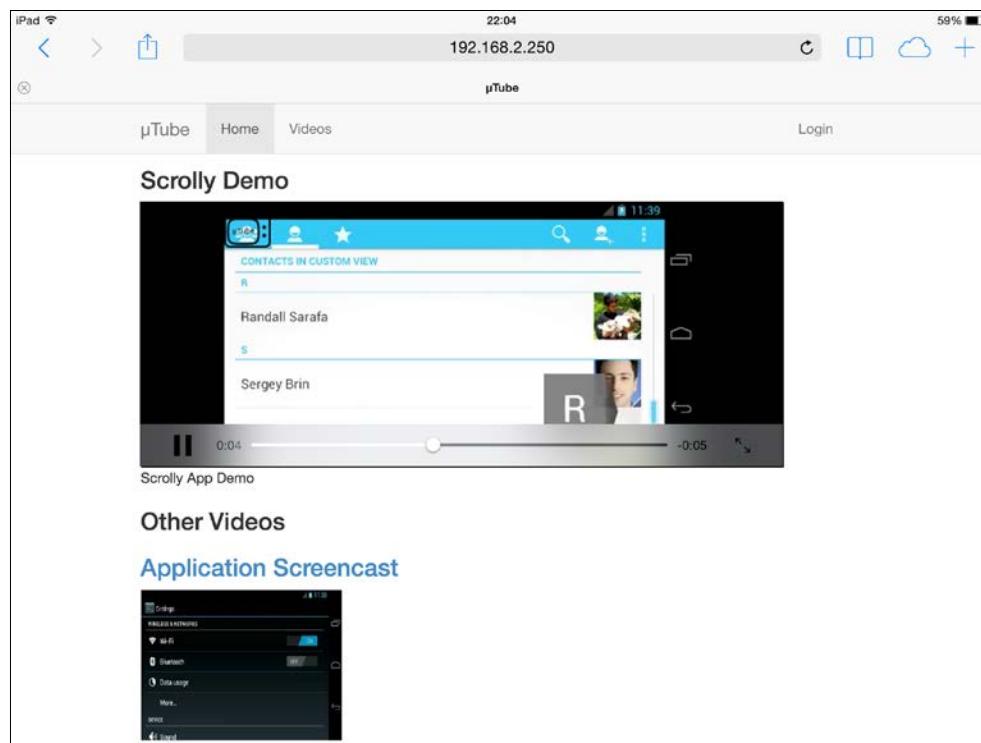
Then we created a `video` element to load the video on it:

```
<video id="video_1" class="video-js vjs-default-skin" controls
preload="none" width="640" height="264"
  data-setup="{}">
  <source src="<%= @video.video.url(:mp4) %>" type='video/mp4' />
</video>
```

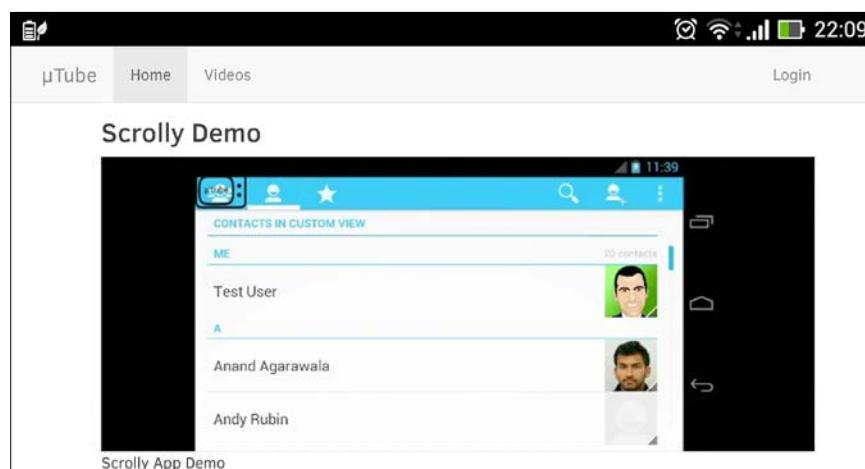
We also created a `helper` method to create our screenshot path and called it in our view.

The `video` element has been introduced in HTML5 as a native HTML tag. By default, it supports MP4 and OGV formats. The work on video support is still in progress but most modern browsers such as Chrome and Firefox support it fully. `video.js` forms a layer on top of the HTML5 element and modifies it to fall back to flash for older browsers and provide more advanced options for video control. It also gives a lot of flexibility to change the skins of the video player.

As part of HTML5 video standard, multiple device compatibility comes inbuilt. We will test our application on devices other than the desktop. The first test is done on iPad Retina, running iOS 7. The video works flawlessly on it as shown in the following screenshot:



We will do the second test on an Android phone, running the Android 4.2.2 Jelly Bean as shown in the following screenshot:



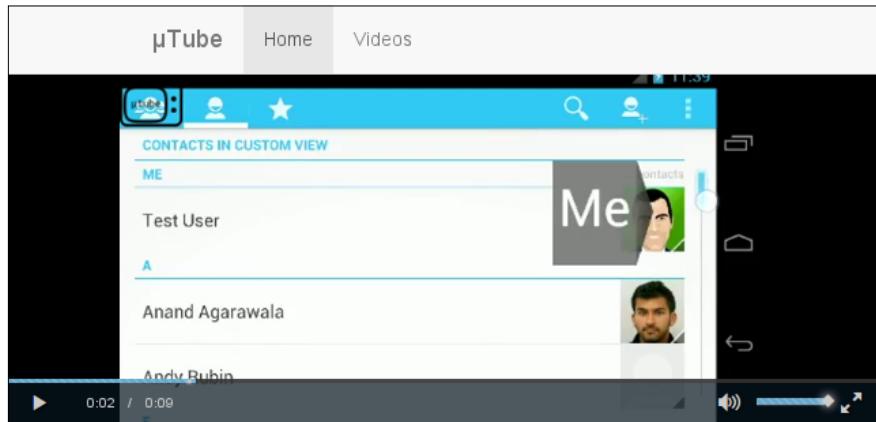
We are using the `friendly_id` gem to generate slugs, so the `params[:id]` will pass the name of slug instead of `id` as the gem modifies the `to_param` method in Rails to call the `name` attribute instead of `id`. In order to access the `id` attribute in the `show` method, we have used the `video` object directly:

```
def show
  @videos = Video.get_other_videos(@video.id)
end
```

We queried our database to get videos other than the current video. Our query omits the `video_id` and finds all videos except the current one. We will use the catch a `nil` exception using `rescue.nil` in order to avoid failures caused due to nil records:

```
videos = Video.where.not(id: video_id) rescue nil
```

Also, we can see the watermark now, the one we created in our previous task, on the top-left side of the screen in the following screenshot:



Caching the content – text and video

In this task, we will look at some of the newer techniques of caching. Russian Doll Caching was introduced in Rails 3.2 and is now used in Rails 4 as the main mechanism of page and fragment caching. It also implements the usage of cache digest. This will lead to effective versioning of cached items even if someone misses out on adding the right version of cache while writing the code.

Engage thrusters

We will now take steps to add specific caching mechanisms to our application:

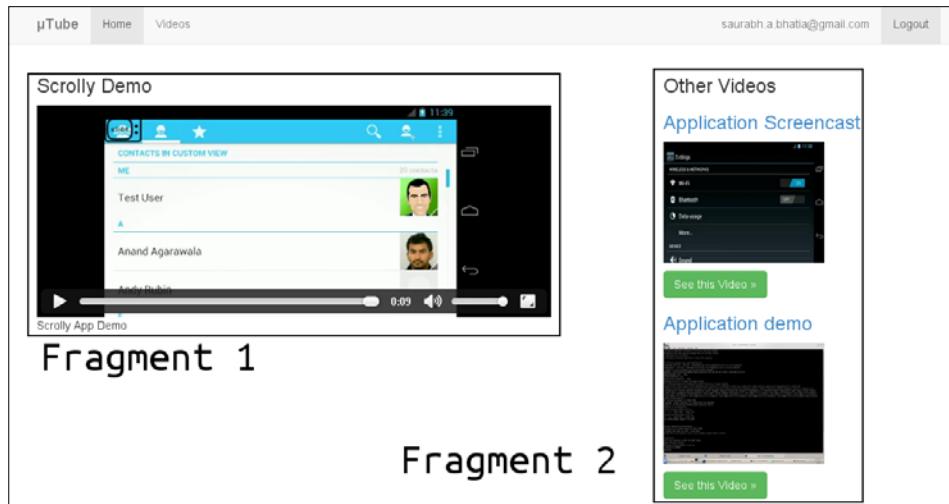
1. In our video model, we will add ActiveRecord's touch method to keep the served content fresh:

```
class Video < ActiveRecord::Base
  belongs_to :user, touch: true
  mount_uploader :video, VideoUploader
  extend FriendlyId
  friendly_id :title, use: :slugged

  def video_thumbnail
    screenshot_path = "#{Rails.root}/app/assets/images/
screenshots/" + "#{self.slug}_#{self.id}.jpg"
    if FileTest.exists?(screenshot_path.to_s)
      @thumbnail = screenshot_path.to_s
    else
      video_file = FFMPEG::Movie.new("#{Rails.root}/public"+self.
video.url(:mp4))
      @thumbnail = video_file.thumbnail(screenshot_path.to_s)
    end
  end

  def self.get_other_videos(video_id)
    videos = Video.where.not(id: video_id) rescue nil
    return videos
  end
end
```

2. We will then cache the various fragments of our view. Our home page has two parts as shown in the following screenshot:



3. In order to keep up with the template changes, we will create versions for our cache. The first segment is the video itself. We will cache `video.title`, `video.description`, and `video` object:

```
app/views/videos/show.html.erb
<% cache ["v1",@video] do %>
  <div class="row">
    <div class="col-lg-8">
      <h3><%= @video.title %></h3>
      <video id="example_video_1" class="video-js vjs-default-skin" controls preload="none" width="640" height="264"
        data-setup="{}">
        <source src="<%= @video.video.url(:mp4) %>" type='video/mp4' />
      </video>
      <br/>
      <p><%= @video.description %></p>
    </div>
  <% end %>
```

4. The second segment for caching is the right-hand side bar where we display the videos. We will cache the `@videos` object, video title, and also the screenshot for all the videos:

```
app/views/videos/show.html.erb
<% cache ["v1",@"videos] do%>
  <div class="col-lg-4">
    <h3>Other Videos</h3>
    <% @videos.each do |video| %>
      <h3><%=link_to video.title, video %></h3>
      <% video.video_screenshot%>
      <p><%= image_tag "screenshots/#{video.slug}_{#{video.id}}.jpg", :width => 200, :height => 150 %></p>
      <p><%= link_to 'See this Video &raquo;', html_safe
        ,video, :class=>"btn btn-success"%></p>
    <%end%>
  </div>
<% end %>
```

5. We will precompile our assets, boot them into production, and reload our page:

```
$ bundle exec rake assets:precompile
```

6. We can see the command-line output, as shown in the following screenshot, after booting into production:

```
I, [2014-04-02T20:58:34.662406 #6298] INFO -- : Started POST "/videos" for 127.0.0.1 at 2014-04-02 20:58:34 +0800
I, [2014-04-02T20:58:34.680902 #6298] INFO -- : Processing by VideosController#create as HTML
I, [2014-04-02T20:58:34.681187 #6298] INFO -- : Parameters: {"utf8"=>"/", "authenticity_token"=>"YpSsZCsp09rzs9N7fevAbNndTwFYIWg7xUVgRvs=", "video"=>{"title"=>"Application Screencast", "description"=>"app screencast", "video"=>#<ActionDispatch::Http::UploadedFile:0x00000001983478 @tempfile=>/tmp/Rack Multipart20140402-6298-10hjxio>, @original_filename="out.ogv", @content_type="video/ogg", @headers="Content-Disposition: form-data; name=\"video[video]\""; filename=\"out.ogv\"\r\nContent-Type: video/ogg\r\n", "video_cache"=>"", "user_id"=>"1"}, "commit"=>"Save"}
[deprecated] I18n.enforce_available_locales will default to true in the future. If you really want to skip validation of your locale you can set I18n.enforce_available_locales = false to avoid this message.
2014-04-02T12:58:34Z 6298 TID-fo3bw INFO: Sidekiq client with redis options {}
I, [2014-04-02T20:58:34.997727 #6298] INFO -- : Redirected to http://localhost:3000/videos/application-screencast
I, [2014-04-02T20:58:34.998411 #6298] INFO -- : Completed 302 Found in 317ms ( ActiveRecord: 64.8ms)
I, [2014-04-02T20:58:35.014639 #6298] INFO -- : Started GET "/videos/application-screencast" for 127.0.0.1 at 2014-04-02 20:58:35 +0800
I, [2014-04-02T20:58:35.016657 #6298] INFO -- : Processing by VideosController#show as HTML
I, [2014-04-02T20:58:35.017087 #6298] INFO -- : Parameters: {"id"=>"application-screencast"}
I, [2014-04-02T20:58:35.024567 #6298] INFO -- : Cache digest for videos/show.html: 7fdd6bbe5flafa9e7dec72e7bb45d3b2
I, [2014-04-02T20:58:35.025527 #6298] INFO -- : Read fragment views/videos/15-20140402125834000000000/expires_in/1800/7fdd6bbe5flafa9e7dec72e7bb45d3b2 (0.2ms)
I, [2014-04-02T20:58:35.027887 #6298] INFO -- : Write fragment views/videos/15-20140402125834000000000/expires_in/1800/7fdd6bbe5flafa9e7dec72e7bb45d3b2 (1.2ms)
I, [2014-04-02T20:58:35.030691 #6298] INFO -- : Read fragment views/videos/14-20140402125314000000000/exp
```

7. In case someone misses the version number in the cache definition, it will lead to an error. Hence, we will make our application version free by removing the versions that are not required:

```
app/views/videos/show.html.erb
<% cache @video do %>
  <div class="row">
    <div class="col-lg-8">
      <h3><%= @video.title %></h3>
      <video id="example_video_1" class="video-js vjs-default-skin" controls preload="none" width="640" height="264" data-setup="{}">
        <source src="<%= @video.video.url(:mp4) %>" type='video/mp4' />
      </video>
      <br/>
      <p><%= @video.description %></p>
    </div>
  <% end %>
  <% cache @videos do %>
    <div class="col-lg-4">
      <h3>Other Videos</h3>
      <% @videos.each do |video| %>
        <h3><%= link_to video.title, video %></h3>
        <% video.video_screenshot%>
      <% end %>
    </div>
  <% end %>
</div>
```

```
<p><%= image_tag "screenshots/#{video.slug}_#{video.id}.jpg", :width => 200, :height => 150 %></p>
<p><%= link_to 'See this Video &raquo;', .html_safe ,video, :class=>"btn btn-success"></p>
<% end %>
</div>
</div>
<% end %>
```

8. We need to prepare our production configuration to load our asset pipeline first:

```
config/environments/production.rb
# Precompile additional assets.
# application.js, application.css, and all non-JS/CSS in app/
assets folder are already added.
config.assets.precompile += ['*.js', '*.css', '*.css.erb']
```

9. After we have prepared our production environment for the cache and asset pipeline, we will boot our production server and render our show page, where we enabled the cache. We can see the cache digest for our videos in the page that will appear in the following screenshot:

```
rwub@rwub:~/rails4-book/book/62940S_Chapter_09/project-9$ rails s -e production
=> Booting Puma
=> Rails 4.0.2 application starting in production on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
Puma 2.8.1 starting...
* Min threads: 0, max threads: 16
* Environment: production
* Listening on tcp://0.0.0.0:3000
I, [2014-04-02T21:03:12.043698 #6861] INFO -- : Started GET "/videos/application-screencast" for 127.0.0.1 at 2014-04-02 21:03:12 +0800
I, [2014-04-02T21:03:12.078202 #6861] INFO -- : Processing by VideosController#show as HTML
I, [2014-04-02T21:03:12.078834 #6861] INFO -- : Parameters: {"id"=>"application-screencast"}
I, [2014-04-02T21:03:12.152084 #6861] INFO -- : Cache digest for videos/show.html: 7fdd6bbe5f1afa9e7dec72e7bb45d3b2
I, [2014-04-02T21:03:12.155837 #6861] INFO -- : Read fragment views/videos/15-20140402125834000000000/expires_in/1800/7fdd6bbe5f1afa9e7dec72e7bb45d3b2 (0.7ms)
I, [2014-04-02T21:03:12.162204 #6861] INFO -- : Read fragment views/videos/14-20140402125314000000000/expires_in/1800/7fdd6bbe5f1afa9e7dec72e7bb45d3b2 (0.5ms)
```

Objective complete – mini debriefing

In this task, we used the Russian Doll caching technique to divide the page into fragments and cache them separately. In order to keep our caching objects free from the complexity of logged in objects, we have cached only those sections that do not need logging in and ones that don't depend on session objects whatsoever. We started by adding the `touch` method to our video-model association:

```
belongs_to :user, touch: true
```

The `touch` method is used to keep the data in the cache fresh. It is particularly useful for associations. So in our use case, whenever a new video is created by a particular user, the cache expires and loads with the details of the new video. Also, when a video has been updated or deleted, the `touch` method automatically resets the cache and updates it.

Then, we added versioned caching to our page fragments:

```
<% cache ["v1", @videos] do%>
<% end %>
```

The cache generates fragments with version numbers as follows:

```
I, [2014-01-07T07:49:18.018543 #10073] INFO -- : Cache digest for
videos/show.html: 17d26e8a6e5adce61cf3a6d90e1eabd5
I, [2014-01-07T07:49:18.020463 #10073] INFO -- : Read fragment views/
v1/videos/6-20140106233228000000000/17d26e8a6e5adce61cf3a6d90e1eabd5
(0.3ms)
I, [2014-01-07T07:49:18.026462 #10073] INFO -- : Write fragment
views/v1/videos/6-20140106233228000000000/17d26e8a6e5adce61cf3a6d90e1
eabd5 (4.2ms)
I, [2014-01-07T07:49:18.030010 #10073] INFO -- : Read fragment views/
v1/videos/5-20140106233152000000000/17d26e8a6e5adce61cf3a6d90e1eabd5
(0.3ms)
```

Then we removed the version numbers in order to avoid pitfalls due to versioning. By default, Rails generates a unique ID and version for cache digests with every render:

```
<% cache @videos do%>
<% end %>
```

The cache now generates fragments without version numbers:

```
I, [2014-01-07T07:33:13.491723 #8193] INFO -- : Read fragment views/
videos/6-20140106233228000000000/8d8424568d7560f7eb85717b3b6e8a71
(0.4ms)
I, [2014-01-07T07:33:13.494620 #8193] INFO -- : Read fragment views/
videos/5-20140106233152000000000/8d8424568d7560f7eb85717b3b6e8a71
(0.3ms)
```

In order to expire or invalidate the cache, we can simply add the `expires_in` option:

```
<% cache [@videos, expires_in: 30.minutes] do%>
```

It should also be noted that we can add `race_condition_ttl` along with our `expires_in` option to avoid something called the dogpile effect. This happens when there are two simultaneous requests and there is a possibility that the cache ID generated for both requests could be the same.

Queuing the job

Video uploading and encoding might sometimes take a lot of time while processing, depending on the size of the video, the kind of Internet connection, and upload bandwidth of the user. In this case, waiting for the video to upload in order to go to other pages could be a very annoying experience for the user. In order to enhance the user experience, we can run our encoding as a background job and make an asynchronous queue of videos in order to schedule and encode them as we do other tasks. We will use Sidekiq to generate queues and manage the background processing; we will connect it to the carrierwave gem.

Prepare for lift off

1. We will start by installing Redis on our machine. We will download the latest copy of the Redis source and build it from its source:

```
$ wget http://download.redis.io/redis-stable.tar.gz  
$ tar xvzf redis-stable.tar.gz  
$ cd redis-stable  
redis-stable$ make
```

We will then start the Redis server:

```
$ redis-server
```

2. This installation works for Linux- and Mac OS X-based systems. For further details on this, you can visit the Redis download page at <http://redis.io/download>.
3. We will test the Redis configuration by pinging the `redis-cli` command:

```
$ redis-cli ping  
PONG
```

4. Add `redis` to our application's `Gemfile` and run `bundle install`:

```
'redis', '>= 3.0.6'  
'redis-namespace', '>= 1.3.1'
```

5. Create an initializer to connect to the local `redis` instance:

```
config/initializers/redis.rb  
  
$redis = Redis.new(:host => 'localhost', :port => 6379)
```

6. We will test the connection to Redis from our Rails console:

```
mutube$ rails c
Loading development environment (Rails 4.0.2)
1.9.3-p327 :001 > $redis
=> #<Redis client v3.0.6 for redis://localhost:6379/0
```

7. Redis is now working within our application, and we can now go ahead with our Sidekiq installation.

Engage thrusters

In the following steps, we will add a queuing mechanism to our application:

1. Add sidekiq to the Gemfile and run bundle install:

```
Gemfile
gem 'sidekiq'
```

2. Once the bundle is successful, we will also need to add a carrierwave extension to run the background job:

```
Gemfile
gem 'carrierwave_backgrounder'
```

3. We will generate the initializer once the gem is successfully installed:

```
mutube$ rails g carrierwave_backgrounder:install
      create config/initializers/carrierwave_backgrounder.rb
```

By default, delayed_job is used as the backend for carrierwave_backgrounder with :carrierwave as the queue name.

To change this, edit config/initializers/carrierwave_backgrounder.rb.

4. In the initializer for carrierwave_backgrounder.rb, we will define sidekiq as the queuing methodology and carrierwave as the default queue name:

```
config/initializers/carrierwave_backgrounder.rb
CarrierWave::Backgrounder.configure do |c|
  #c.backend :delayed_job, queue: :carrierwave
  # c.backend :resque, queue: :carrierwave
  c.backend :sidekiq, queue: :carrierwave
  # c.backend :girl_friday, queue: :carrierwave
  # c.backend :sucker_punch, queue: :carrierwave
  # c.backend :qu, queue: :carrierwave
  # c.backend :qc
end
```

5. We will include the CarrierWave::Backgrounder module in our uploader:

```
app/uploaders/video_uploader.rb
class VideoUploader < CarrierWave::Uploader::Base
  include CarrierWave::MimeTypes
  include CarrierWave::Video
  include ::CarrierWave::Backgrounder::Delay
  storage :file
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.
id}"
  end
  DEFAULTS = {
    watermark: {
      path: Rails.root.join('mutube.png')
    }
  }
  version :mp4 do
    process :encode
  end
  version :screenshot do
    process :screenshot
  end
  def encode
    encode_video(:mp4, DEFAULTS) do |movie, params|
      if movie.height < 720
        params[:watermark][:path] = Rails.root.join('mutube.png')
      end
    end
  end
  def extension_white_list
    %w(mp4 ogv avi)
  end
end
```

6. We will load this module onto our model using a method called `process_in_background` and call our uploader in it:

```
app/models/video.rb
class Video < ActiveRecord::Base
  belongs_to :user, touch: true
  mount_uploader :video, VideoUploader
  extend FriendlyId
  friendly_id :title, use: :slugged
  process_in_background :video
```

```
def video_screenshot
  screenshot_path = "#{Rails.root}/app/assets/images/
screenshots/+"#{self.slug}_#{self.id}.jpg"
  if FileTest.exists?(screenshot_path.to_s)
    @screenshot = screenshot_path.to_s
  else
    video_file = FFMPEG::Movie.new("#{Rails.root}/public"+self.
video.url(:mp4))
    @screenshot = video_file.screenshot(screenshot_path.to_s)
  end
end
def self.get_other_videos(video_id)
  videos = Video.where.not(id: video_id) rescue nil
  return videos
end
```

7. We will now load our console and make a test call on our Sidekiq method to check whether everything is fine or not:

```
Loading development environment (Rails 4.0.2)
1.9.3-p327 :001 > Sidekiq::Client.registered_workers
2014-01-05T10:56:01Z 20119 TID-2dbz4 INFO: Sidekiq client using
redis://localhost:6379/0 with options {}
=> []
```

8. Looks good! So we can now fire up our Sidekiq server:

```
$ bundle exec sidekiq -q carrierwave,5 default
2014-01-06T23:10:41Z 4735 TID-5rkyo INFO: Booting Sidekiq 2.14.0
using redis://localhost:6379/0 with options {}
2014-01-06T23:10:41Z 4735 TID-5rkyo INFO: Running in ruby
1.9.3p327 (2012-11-10 revision 37606) [x86_64-linux]
2014-01-06T23:10:41Z 4735 TID-5rkyo INFO: See LICENSE and the
LGPL-3.0 for licensing details.
2014-01-06T23:10:41Z 4735 TID-5rkyo INFO: Starting processing, hit
Ctrl-C to stop
2014-01-06T23:10:43Z 4735 TID-c6ffw CarrierWave::Workers::ProcessA
sset JID-c565e44604416585b8a01a8e INFO: start
2014-01-06T23:10:43Z 4735 TID-c8hgc CarrierWave::Workers::ProcessA
sset JID-7491d6a930d19b5f593197d8 INFO: start
2014-01-06T23:10:44Z 4735 TID-c6ffw CarrierWave::Workers::ProcessA
sset JID-c565e44604416585b8a01a8e INFO: done: 1.044 sec
2014-01-06T23:10:44Z 4735 TID-c8hgc CarrierWave::Workers::ProcessA
sset JID-7491d6a930d19b5f593197d8 INFO: done: 1.04 sec
```

9. We will now try to upload a video and see how the process looks in our console:

```
rwub@rwub:~/rails4-book/book/62940S_Chapter_09/project-9$ bundle exec sidekiq -q carrierwave,5 default
2014-04-02T13:14:37Z 8184 TID-5ayoc INFO: Booting Sidekiq 2.17.7 with redis options {}
2014-04-02T13:14:37Z 8184 TID-5ayoc INFO: Running in ruby 2.1.1p76 (2014-02-24 revision 45161) [x86_64-linux]
2014-04-02T13:14:37Z 8184 TID-5ayoc INFO: See LICENSE and the LGPL-3.0 for licensing details.
2014-04-02T13:14:37Z 8184 TID-5ayoc INFO: Starting processing, hit Ctrl-C to stop
2014-04-02T13:14:49Z 8184 TID-al608 CarrierWave::Workers::ProcessAsset JID-8a4fcb3927146084d31cdf19 INFO:
  start
I, [2014-04-02T21:14:49.329591 #8184] INFO -- : Running transcoding...
ffmpeg -y -i /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444489-8184-9674/mp4_out.ogv -vcodec libx264 -s 640x364 -strict experimental -preset slow -g 30 -vf "movie=/home/rwub/rails4-book/book/62940S_Chapter_09/project-9/mutube.png [logo]; [in][logo] overlay=[out]" -aspect 1.758 2417582417582 /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444489-8184-9674/tmpfile.mp4

I, [2014-04-02T21:14:52.637709 #8184] INFO -- : Transcoding of /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444489-8184-9674/mp4_out.ogv to /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444489-8184-9674/tmpfile.mp4 succeeded

2014-04-02T13:14:52Z 8184 TID-al608 CarrierWave::Workers::ProcessAsset JID-8a4fcb3927146084d31cdf19 INFO:
  done: 3.46 sec
2014-04-02T13:15:03Z 8184 TID-al608 CarrierWave::Workers::ProcessAsset JID-ad6b9bd2a4859f64fb4a32cf INFO:
  start
I, [2014-04-02T21:15:03.124079 #8184] INFO -- : Running transcoding...
ffmpeg -y -i /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444503-8184-9483/mp4_out.ogv -vcodec libx264 -s 640x364 -strict experimental -preset slow -g 30 -vf "movie=/home/rwub/rails4-book/book/62940S_Chapter_09/project-9/mutube.png [logo]; [in][logo] overlay=[out]" -aspect 1.758 2417582417582 /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444503-8184-9483/tmpfile.mp4

I, [2014-04-02T21:15:08.845149 #8184] INFO -- : Transcoding of /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444503-8184-9483/mp4_out.ogv to /home/rwub/rails4-book/book/62940S_Chapter_09/project-9/public/uploads/tmp/1396444503-8184-9483/tmpfile.mp4 succeeded
```

10. We will also enable the web interface for Sidekiq. For this, we need `sinatra`. In our `Gemfile`, add `sinatra` and run `bundle install`:

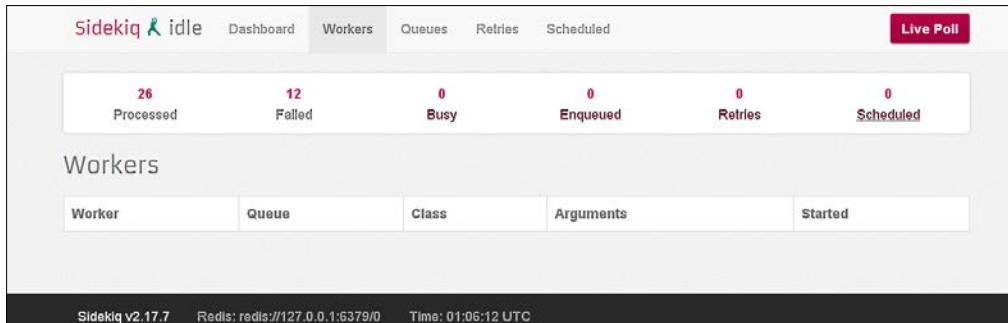
Gemfile

```
gem 'sinatra', '>= 1.3.0', :require => nil
```

11. Mount the Sidekiq route in our application routes:

```
config/routes.rb
require 'sidekiq/web'
MuTube::Application.routes.draw do
  devise_for :users
  get "home/index"
  resources :videos
  mount Sidekiq::Web => '/sidekiq'
  root 'videos#index'
end
```

12. Reboot the server and browse to `http://localhost:3000/sidekiq/`. We can now see the Sidekiq job management dashboard in the following screenshot:

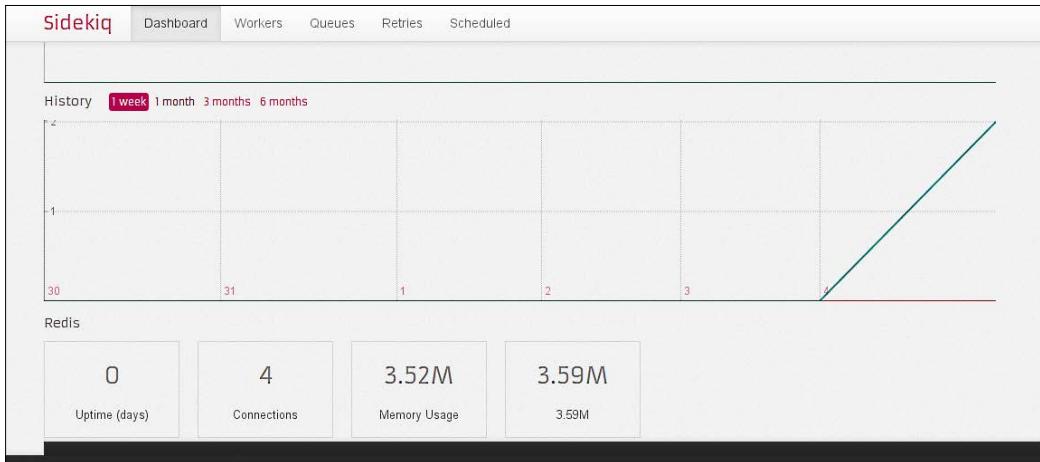


Objective complete – mini debriefing

Before we began the task, we prepared our system to run Redis. This is a dependency to run Sidekiq. It is worth noting that most of the job queues use Redis as a choice of persistence. This is because Redis is fast, easy to manage, and is document-oriented. Sidekiq has other popular alternatives as well, for example, resque and delayed job. The reasons we chose Sidekiq over the other two were as follows:

- ▶ Delayed job has no management dashboard
- ▶ Resque is known to be more memory inefficient than Sidekiq

Though we need to make sure our code is threadsafe with Sidekiq, as it inherently does not protect itself from non-threadsafe objects, it still is good enough for most tasks as it occupies much less memory than its peers. The following is an example of the Sidekiq dashboard, which even provides reports for jobs:



We first enabled sidekiq with the carrierwave queue.

```
c.backend :sidekiq, queue: :carrierwave
```

Then, we loaded the module for queues in our uploader:

```
include ::CarrierWave::Backgrounder::Delay
```

Furthermore, we enabled a background job in the model. So now when we upload a video, it is immediately sent to a queue. Until then, we can proceed to do other tasks as the video is being encoded and worked on in the backend. We do have to make sure, however, that the server for Sidekiq needs to be up all the time. The other purposes of using queues could be mailing newsletters or reindexing solr.

Mission accomplished

We have successfully created an app where we can upload and encode a video, then display it. Let's have a quick recap of what we did.

Some of the areas we covered in this project are as follows:

- ▶ We created a Rails app, video model, and uploader. We also added methods to create slugs using the `friendly_id` gem.
- ▶ We restricted the formats of videos to be uploaded.
- ▶ We installed `ffmpeg` and its dependencies.
- ▶ We used customized `carrierwave-video` to suit our needs for encoding the video. We transcoded the videos to the MP4 format.
- ▶ In order to display the video, we used `video.js`.
- ▶ We made sure it works on multiple devices and platforms.
- ▶ We used Russian Doll caching to cache our videos, screenshots, and text.
- ▶ We used Sidekiq to create and manage queues for multiple video uploads.

We are using `ffmpeg` that is compiled from the source. If we are using it for production, we need to be very sure about fixing a version for a long time. This is because command-line flags and encoding filenames in `ffmpeg` change very often, and there are several deprecations between versions. To check the current configuration of `ffmpeg`, we will run the following code:

```
$ ffmpeg
ffmpeg version git-2014-01-01-07728a1 Copyright (c) 2000-2013 the
FFmpeg developers
  built on Jan 1 2014 20:16:31 with gcc 4.8 (Ubuntu/Linaro
4.8.1-10ubuntu9)
```

```
configuration: --prefix=/home/rwub/ffmpeg_build --extra-cflags=-I/home/rwub/ffmpeg_build/include --extra-ldflags=-L/home/rwub/ffmpeg_build/lib --bindir=/home/rwub/bin --extra-libs=ldl --enable-gpl --enable-libass --enable-libfdk-aac --enable-libmp3lame --enable-libopus --enable-libtheora --enable-libvorbis --enable-libvpx --enable-libx264 --enable-nonfree --enable-x11grab --enable-libfaac libavutil      52. 59.100 / 52. 59.100
libavcodec     55. 47.100 / 55. 47.100
libavformat    55. 22.102 / 55. 22.102
libavdevice    55.  5.102 / 55.  5.102
libavfilter     4.  0.103 /  4.  0.103
libswscale      2.  5.101 /  2.  5.101
libswresample   0. 17.104 /  0. 17.104
libpostproc    52.  3.100 / 52.  3.100
Hyper fast Audio and Video encoder
usage: ffmpeg [options] [[infile options] -i infile]... {[outfile options] outfile}...
Use -h to get full help or, even better, run 'man ffmpeg'
```

MP4 encoding is a tricky thing using `ffmpeg`. By default, MP4 hinting is not set (hinting allows the video to be streamed as soon as it completes to load by setting a flag). This disables the autoplay completely, as it looks to download the entire file before it starts playing. MP4Box or Nginx MP4 streaming can be used for this purpose. We need to make sure of the following factors:

- ▶ Make sure the file size limit is defined and is enough for videos to be uploaded
- ▶ Allow incoming files while uploading

Hotshot challenges

We created a fully functional video platform that can be used to upload and manage videos. We can extend it further to make it even cooler:

- ▶ Allow encoding to the OGV and Theora format of the uploaded videos
- ▶ Expire the cache key after 5 minutes
- ▶ Add custom skins to `video.js`
- ▶ Enable automatic retry in case the job fails
- ▶ If the file size of video is very small, then bypass the background job

Project 10

A Rails Engines-based E-Commerce Platform

Rails provides an effective way to extend the functionality of applications in a plug-and-play fashion. This is called **Rails Engines**. Earlier Rails versions had engines and plugins (which are located at `app/vendor/plugins`), but Rails 4 has completely deprecated the use of plugins in Rails apps. Compared to plugins, engines are cleaner in terms of their definition, have a proper testing structure, and can be more easily customized.

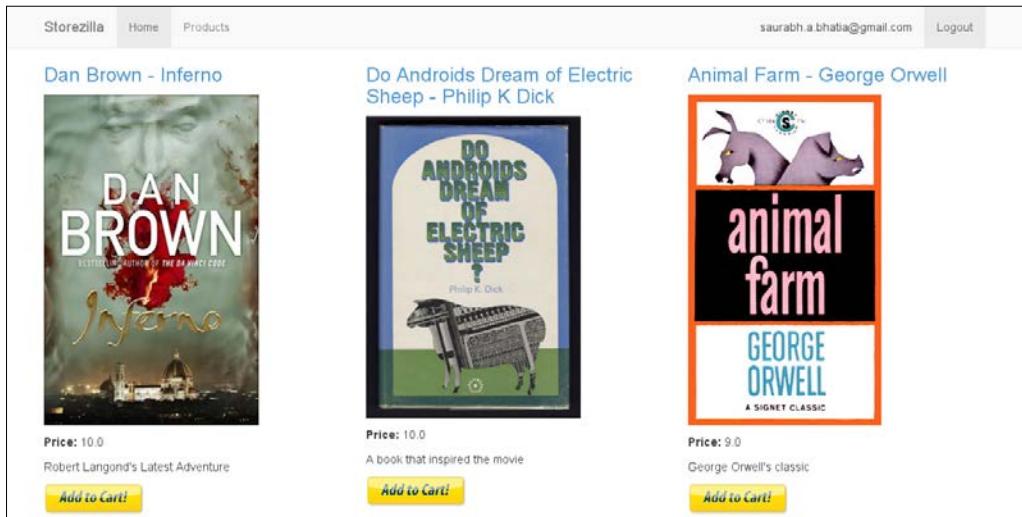
Mission briefing

In this section, we will create a Rails engine to generate an e-commerce application. Once ready, we will add the entire application as a gem and mount it on the main application. As soon as we do this, we will get all the basic features of a shopping cart application. This will allow users to maintain the application and maintain a collection of multiple modules. The application will have different moving parts that need to be upgraded on a frequent basis, as and when they are updated by their respective maintainers.

Why is it awesome?

There are several examples of successful, feature-packed Rails engines such as Devise, Spree Commerce, and LocomotiveCMS. These engines have given users an easy way to incorporate really advanced functionality such as an authentication system, a fully featured e-commerce engine, and a content management system tucked inside a Ruby gem.

The following screenshot shows us what the designed application looks like:



At the end of these tasks, we will have a Rails engine that can be mounted on a Rails application.

Your Hotshot objectives

While building this application, we will go through the following tasks:

- ▶ Creating a category and product listing
- ▶ Creating a shopping cart and an **add to cart** feature
- ▶ Packaging the engine as a gem
- ▶ Mounting the engine on a blank Rails application
- ▶ Customizing and overriding the default classes

Mission checklist

We need the following software installed on the system before we start with our mission:

- ▶ Ruby 1.9.3 / Ruby 2.0.0
- ▶ Rails 4.0+
- ▶ MongoDB 2.4
- ▶ Devise
- ▶ Bootstrap 3.0
- ▶ Git
- ▶ jQuery

Creating a category and product listing

In the first task, we will deal with the creation of a Rails engine. We will create a product and category to list our products as we are creating an e-commerce engine. We will see how to add a carrierwave uploader for uploading product images inside the engine and add it as a dependency to our application. At the end of this, we will understand why we selected a mountable Rails engine instead of a full Rails engine.

Engage thrusters

We will first create the backbone of our Rails engine by performing the steps:

1. We will generate a mountable engine as opposed to a full Rails engine.

```
$rails plugin new ecom --mountable --O
      create
      create README.rdoc
      create Rakefile
      create ecom.gemspec
      create MIT-LICENSE
      create .gitignore
      create Gemfile
      create app
      create app/controllers/ecom/application_controller.rb
      create app/helpers/ecom/application_helper.rb
      create app/mailers
      create app/models
      create app/views/layouts/ecom/application.html.erb
      create app/assets/images/ecom
      create app/assets/images/ecom/.keep
      create config/routes.rb
      create lib/ecom.rb
      create lib/tasks/ecom_tasks.rake
      create lib/ecom/version.rb
      create lib/ecom/engine.rb
      create app/assets/stylesheets/ecom/application.css
      create app/assets/javascripts/ecom/application.js
      create bin
      create bin/rails
      create test/test_helper.rb
      create test/ecom_test.rb
```

```
append Rakefile
create test/integration/navigation_test.rb
vendor_app test/dummy
run bundle install
Fetching gem metadata from https://rubygems.org/.....
Fetching gem metadata from https://rubygems.org/..
Resolving dependencies...
Using rake (10.1.1)
Using i18n (0.6.9)
Using minitest (4.7.5)
Using multi_json (1.8.2)
Using atomic (1.1.14)
Using thread_safe (0.1.3)
Using tzinfo (0.3.38)
Using activesupport (4.0.2)
Using builder (3.1.4)
Using erubis (2.7.0)
Using rack (1.5.2)
Using rack-test (0.6.2)
Using actionpack (4.0.2)
Using mime-types (1.25.1)
Using polyglot (0.3.3)
Using treetop (1.4.15)
Using mail (2.5.4)
Using actionmailer (4.0.2)
Using activemodel (4.0.2)
Using activerecord-deprecated_finders (1.0.3)
Using arel (4.0.1)
Using activerecord (4.0.2)
Using bundler (1.3.5)
Using thor (0.18.1)
Using railties (4.0.2)
Using hike (1.2.3)
Using tilt (1.4.1)
Using sprockets (2.10.1)
Using sprockets-rails (2.0.1)
Using rails (4.0.2)
Using ecom (0.0.1) from source at /home/rwub/rails4-book/
book/6294OS_Chapter_10/ecom
ecom at /home/rwub/rails4-book/book/6294OS_Chapter_10/ecom did not
have a valid gemspec.
This prevents bundler from installing bins or native extensions,
but that may not affect its functionality.
The validation message from Rubygems was:
```

```
"FIXME" or "TODO" is not an author
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is
installed.
```

2. As we have skipped ActiveRecord, we need an ORM, so, we will add Mongoid to our Gemfile and bundle install:

```
Gemfile
```

```
gem 'mongoid', github: 'mongoid/mongoid'
ecom$ bundle install
```

3. We will not run mongoid:config in the Rails engine; we will do this after this engine is installed in an application.
4. In order to use mongoid to generate our models, we need to add it as a module dependency in our Rails binary. Currently, it looks like the following code:

```
bin/rails
```

```
#!/usr/bin/env ruby
# This command will automatically be run when you run "rails" with
Rails 4 gems installed from the root of your application.

ENGINE_ROOT = File.expand_path('../..', __FILE__)
ENGINE_PATH = File.expand_path('../..lib/ecom/engine', __FILE__)

# Set up gems listed in the Gemfile.

ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../Gemfile', __
FILE__)

require 'bundler/setup' if File.exist?(ENV['BUNDLE_GEMFILE'])

require "rails/all"
require 'rails/engine/commands'
```

5. We will modify "rails/all" to load all the modules separately and especially load mongoid. We will also load rubygems from the gemspec file directly onto the bin/rails file.

```
bin/rails
```

```
#!/usr/bin/env ruby
# This command will automatically be run when you run "rails" with
Rails 4 gems installed from the root of your application.

ENGINE_ROOT = File.expand_path('../..', __FILE__)
ENGINE_PATH = File.expand_path('../..lib/ecom/engine', __FILE__)

# Set up gems listed in the Gemfile.
```

```
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../..../Gemfile', __FILE__)

require 'rubygems'

require 'bundler/setup' if File.exist?(ENV['BUNDLE_GEMFILE'])

require "action_controller/railtie"
require "action_mailer/railtie"
require "sprockets/railtie"
require "rails/test_unit/railtie"
require 'rails/engine/commands'
require "mongoid"
```

6. In Rails 4, `active_resource/railties` is not required, so we will have to make sure the following line is not included:

```
require "active_resource/railtie"
```

7. We will also add `mongoid` as a dependency in our `gemspec` file:

```
ecom/com.gemspec

s.add_dependency "rails", "~> 4.1.0.rc1"
s.add_dependency "mongoid", "4.0.0.beta1"
```

8. Generate `rails scaffold` for the products. This will create a model, view, and controller under the `ecom` namespace as shown in the following code:

```
ecom$ rails g scaffold product name:string description:string
base_price:float sku:string
      invoke  mongoid
      create    app/models/com/product.rb
      invoke  test_unit
      create    test/models/com/product_test.rb
      create    test/fixtures/com/products.yml
      invoke  resource_route
      route    resources :products
      invoke  scaffold_controller
      create    app/controllers/com/products_controller.rb
      invoke  erb
      create    app/views/com/products
      create    app/views/com/products/index.html.erb
      create    app/views/com/products/edit.html.erb
      create    app/views/com/products/show.html.erb
      create    app/views/com/products/new.html.erb
      create    app/views/com/products/_form.html.erb
      invoke  test_unit
      create    test/controllers/com/products_controller_test.rb
```

```
invoke  helper
create  app/helpers/ecom/products_helper.rb
invoke  test_unit
create  test/helpers/ecom/products_helper_test.rb
invoke  assets
invoke  js
create  app/assets/javascripts/ecom/products.js
invoke  css
create  app/assets/stylesheets/ecom/products.css
invoke  css
create  app/assets/stylesheets/scaffold.css
```

9. At this point, we will also set up a mechanism to create search-friendly URLs also known as slugs for our products:

```
Gemfile
```

```
gem 'mongoid_slug', "3.2"
```

10. In order to make it work on the product model, we will have to include the module for Mongoid::Slug. We will tell the module to use names to create the slug and enable the history feature in the URL.

```
module Ecom
  class Product
    include Mongoid::Document
    include Mongoid::Slug
    field :name, type: String
    field :description, type: String
    field :base_price, type: Float
    field :sku, type: String
    slug :name, history: true
  end
end
```

11. Likewise, we will create a model for categories too.

```
ecom$ rails g model category title:string
invoke mongoid
create app/models/ecom/category.rb
invoke test_unit
create test/models/ecom/category_test.rb
create fixtures/ecom/categories.yml
```

12. We will associate categories and products:

```
app/models/ecom/category.rb

module Ecom
  class Category
    include Mongoid::Document
    field :title, type: String

    has_many :products
  end
end
```

13. We will also associate the product with the category.

```
app/models/ecom/product.rb

Module Ecom
  class Product
    include Mongoid::Document
    include Mongoid::Slug

    field :name, type: String
    field :description, type: String
    field :base_price, type: Float
    field :sku, type: String
    field :category_id, type: String

    slug :name, history: true
    belongs_to :category
  end
end
```

14. We will follow the same steps to include the carrierwave uploader as we did in our previous projects. We will run the generator for carrierwave as follows:

```
ecom$ rails g uploader image
      create  app/uploaders/ecom/image_uploader.rb
```

15. Note that the uploader is created in the namespace for the ecom/image_uploader.rb plugin.

16. In order to take our plugin for a test drive, we will directly navigate to the test/dummy folder, where a dummy application has been created for us when we generated a new mountable plugin.

17. We will run bundle install and configure our database as per Mongoid:

```
ecom/test/dummy:~/rails g mongoid:config
```

18. This will generate the mongoid config file. We will then start the server. However, we will receive the following error:

```
Unable to autoload constant Ecom::ImageUploader, expected /home/rwub/.rvm/gems/ruby-1.9.3-p327/bundler/gems/ecom-ed9e6082e731/app/uploaders/ecom/image_uploader.rb to define it
```

19. This is because carrierwave, by default, creates the engine namespace folder and places the uploader file in it, but does not modify the uploader file with the module name.

```
app/uploaders/ecom/image_uploader.rb
# encoding: utf-8
module Ecom
  class ImageUploader < CarrierWave::Uploader::Base

    # Choose what kind of storage to use for this uploader:
    storage :file

    # Override the directory where uploaded files will be stored.
    # This is a sensible default for uploaders that are meant to be
    mounted:
    def store_dir
      "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
    end
  end
end
```

Objective complete – mini debriefing

In this task, we created a mountable Rails engine. There are two types of Rails engines available:

- ▶ A full engine
- ▶ A mountable engine

A full engine is a tightly coupled application, which works as a direct augmentation to the existing Rails application. This happens because it shares the classes across the application once included due to the lack of a different namespace. As shown in the following code, a full Rails engine initializer has a regular engine initializer rule:

```
ecom/lib/ecom/engine.rb
module Ecom
  class Engine < ::Rails::Engine
  end
end
```

In our case, we are using a mountable engine so the initializer will have a method called `isolate_namespace`. This method will separate model, views, controllers, and all methods into a namespace called `Ecom`.

```
module Ecom
  class Engine < ::Rails::Engine
    isolate_namespace Ecom
  end
end
```

Everything we see here is included with that namespace.

```
ecom/app/controllers/ecom$ ls
application_controller.rb  categories_controller.rb  products_
controller.rb
```

Because of the namespace, `application_controller` is added to the controller as a dependency before it is extended to make the `ActionController::Base` class available to all the controllers.

```
ecom/app/controllers/ecom/products_controller.rb
require_dependency "ecom/application_controller"

module Ecom
  class ProductsController < ApplicationController
  end
end
```

The main purpose of our engine is to augment an existing application and avoid conflicts with an application's existing model and controller classes. Hence, we decided to go for an `isolate_namespace` mountable engine.

While the plugin was being generated, we saw that a full application to test drive the engine was also created inside the `test` folder. We, however, need to add the database config files in order to run it.

In order to use mongoid inside the Rails engine, we had to manually include the `mongoid` module and hence the other Rails modules in it. This is because `ActiveRecord` is loaded in the `rails/all` module inclusion by default. Hence, we explicitly require specific railties that include `mongoid`. We also added a method to load `rubygems` inside our Rails `bin` file. We also added `mongoid` as a dependency to our Rails engine. `Railtie` is the core of the Rails framework. `ActiveRecord`, `ActionController`, and `ActionMailer` are all examples of `Railtie` and are responsible for initializing themselves. `Railtie` is essential when the component needs to communicate with the Rails framework at the time of boot or even after that.

We created a method to generate search-friendly URLs using the `mongoid-slug` gem. We defined the `name` field to create the slug and enabled history to retain the URLs even after they have been updated.

Creating a shopping cart and an Add to Cart feature

A shopping cart is the most important feature of an e-commerce application. We need to create a temporary session object in order to store the value of items in the cart. The standard terminology for products that have been added to the cart is line items. When a user successfully checks out, line items get transitioned into orders, and they generally live between sessions. They are also dependent on the completion of the transaction. Once the transaction is completed or the session is cleared, the line items are deleted.

Prepare for lift off

We will install devise and generate a model for the user as follows:

```
ecom$ rails g devise:install
      create config/initializers/devise.rb
      create config/locales/devise.en.yml

ecom$ rails g devise user
```

This will generate the following route in our engine's routes:

```
config/routes.rb

devise_for :users, :class_name => "Ecom::User"
```

Engage thrusters

We will create a checkout process in this task, as shown in the following steps:

1. We will first add devise to the application. However, we need to modify a few things in order for it to function seamlessly inside an engine. First, modify the routes:

```
app/config/routes.rb
devise_for :users, {
  :class_name => "Ecom::User",
  module: :devise
}
```

2. Add a router name inside the devise initializer. We will also need to uncomment `secret_key` as follows:

```
app/config/initializer/devise.rb
Devise.setup do |config|
  config.secret_key =
'1c17867bf2d8e469ed713b1249eab0f87c918e0e09b265be6a0ed8bc
01c8f0ebd192387418d60542c96ad42b61fdc8a167ec5843f6cd94e9d
66ee39b33ede703'
  config.parent_controller = 'ActionController::Base'
  config.mailer_sender = 'please-change-me-at-config-initializers-
devise@example.com'
  require 'devise/orm/mongoid'
  config.case_insensitive_keys = [ :email ]
  config.strip_whitespace_keys = [ :email ]
  config.skip_session_storage = [:http_auth]
  config.stretches = Rails.env.test? ? 1 : 10
  config.reconfirmable = true
  config.password_length = 8..128
  config.reset_password_within = 6.hours
  config.sign_out_via = :delete
  config.router_name = :ecom
end
```

3. Finally, add devise as a gem dependency as follows:

```
ecom/com.gemspec
s.add_dependency "rails", "~> 4.1.0.rc1"
s.add_dependency "mongoid", "4.0.0.beta1"
s.add_dependency "devise"
```

4. We will generate a model for line items as follows:

```
$ rails g model line_item product_id:string price:float
      invoke mongoid
      create app/models/com/line_item.rb
      invoke test_unit
      create test/models/com/line_item_test.rb
      create test/fixtures/com/line_items.yml
```

5. We will also generate a model called purchases as shown in the following code. This model stores the value of orders that are generated as soon as the transaction is complete:

```
rails g model purchase user_id:string checked_out_at:time total_
price:float
      invoke mongoid
```

```
create    app/models/ecom/purchase.rb
invoke    test_unit
create    test/models/ecom/purchase_test.rb
create    test/fixtures/ecom/purchases.yml
```

6. First, we will create two associations: the first one between line items and purchases, and the second one between line items and products. This is because a product data is imported into a line item, and upon a successful checkout, the line item is then transformed into a purchase. So, a line item belongs to product and a purchase has many line items.

```
app/model/ecom/line_item.rb
module Ecom
  class LineItem
    include Mongoid::Document
    include Mongoid::Timestamps

    field :purchase_id, type: String
    field :product_id, type: String
    field :price, type: Float

    belongs_to :purchase
    belongs_to :product
  end
end

app/model/ecom/product.rb

module Ecom
  class Product
    include Mongoid::Document
    include Mongoid::Slug

    field :name, type: String
    field :description, type: String
    field :base_price, type: Float
    field :sku, type: String

    slug :name, history: true
    belongs_to :category
    has_many :line_items
    mount_uploader :image, ImageUploader
  end
end
```

```
app/models/ecom/purchase.rb

module Ecom
  class Purchase
    include Mongoid::Document
    include Mongoid::MultiParameterAttributes
    include Mongoid::Timestamps

    field :user_id, type: String
    field :checked_out_at, type: DateTime
    field :total_price, type: Float

    has_many :line_items, :dependent => :destroy
    belongs_to :user
  end
end
```

7. Finally, we will add an association between the user and the purchase:

```
app/models/ecom/user.rb

module Ecom
  class User
    include Mongoid::Document
    include Mongoid::Timestamps

    # Include default devise modules. Others available are:
    # :confirmable, :lockable, :timeoutable and :omniauthable
    devise :database_authenticatable, :registerable,
           :recoverable, :rememberable, :trackable, :validatable

    ## Database authenticatable
    field :email,             :type => String, :default => ""
    field :encrypted_password, :type => String, :default => ""

    ## Recoverable
    field :reset_password_token,   :type => String
    field :reset_password_sent_at, :type => Time

    ## Rememberable
    field :remember_created_at, :type => Time

    ## Trackable
    field :sign_in_count,       :type => Integer, :default => 0
    field :current_sign_in_at,  :type => Time
    field :last_sign_in_at,     :type => Time
    field :current_sign_in_ip,  :type => String
    field :last_sign_in_ip,     :type => String
```

```
## Confirmable
# field :confirmation_token,    :type => String
# field :confirmed_at,          :type => Time
# field :confirmation_sent_at,  :type => Time
# field :unconfirmed_email,     :type => String # Only if using
reconfirmable

## Lockable
# field :failed_attempts,      :type => Integer, :default => 0 # Only
if lock strategy is :failed_attempts
# field :unlock_token,         :type => String # Only if unlock
strategy is :email or :both
# field :locked_at,            :type => Time

has_many :purchases, :dependent => :destroy
end
end
```

8. We will now create a cart controller to display the cart and carry out certain functions such as checkout:

```
$ rails g controller cart show
      create  app/controllers/ecom/cart_controller.rb
      route   get "cart/show"
      invoke  erb
      create   app/views/ecom/cart
      create   app/views/ecom/cart/show.html.erb
      invoke  test_unit
      create   test/controllers/ecom/cart_controller_test.rb
      invoke  helper
      create   app/helpers/ecom/cart_helper.rb
      invoke  test_unit
      create   test/helpers/ecom/cart_helper_test.rb
      invoke  assets
      invoke  js
      create   app/assets/javascripts/ecom/cart.js
      invoke  css
      create   app/assets/stylesheets/ecom/cart.css
```

9. In order to create the line item, we will add a class method in our line_item model as follows:

```
app/models/line_item.rb

def self.make_items(purchase_id, product_id, price)

  LineItem.create(purchase_id: purchase_id, product_id:
product_id, price: price)

end
```

10. While shopping, a user can add multiple products to the cart. Every time the user adds an item, the price is recalculated, as shown in the following code:

```
app/models/ecom/purchase.rb

def recalculate_price!
    self.total_price = line_items.inject(0.0){|sum, line_item| sum
+= line_item.price }

    save!
end
```

11. We will now create methods to add and remove line items from the cart and add a way to pass the objects to the checkout page:

```
app/controllers/ecom/cart_controller.rb
require_dependency "ecom/application_controller"

module Ecom
  class CartController < ApplicationController
    before_filter :authenticate_user!
    before_action :get_cart_value

    def add
      @cart.save
      session[:cart_id] = @cart.id
      product = Product.find(params[:id])
      item = LineItem.new
      item.make_items(@cart.id, product.id, product.base_price)
      @cart.recalculate_price!
      flash[:notice] = "Product Added to Cart"
      redirect_to cart_path
    end

    def remove
      item = @cart.line_items.find(params[:id])
      item.destroy
      @cart.recalculate_price!
      flash[:notice] = "Product Deleted from Cart"
      redirect_to cart_path
    end

    protected

    def get_cart_value
      if session[:cart_id].nil?
        @cart = Purchase.create
        session[:cart_id] = @cart.id
      end
    end
  end
end
```

```
    @cart
  else
    @cart = Purchase.find(session[:cart_id])
  end
end

end
end
```

12. We will display all the items in the cart on the cart page:

```
app/views/ecom/cart/show.html.erb
<h1>Shopping Cart</h1>

<% unless @cart.line_items.any? %>
  <p>You don't have any items in your cart. <%= link_to "Go Add Some", products_path %>
<% end %>

<table width="100%">
  <tr>
    <th>Product</th>
    <th>Price</th>
  </tr>
  <% for line_item in @cart.line_items %>
    <tr>
      <td><%= line_item.product.name %></td>
      <td><%= number_to_currency line_item.price %></td>
      <td><%= link_to "Remove", remove_from_cart_path(line_item), :method => :post %></td>
    </tr>
  <% end %>
  <tr>
    <td>Total:</td>
    <td><%= number_to_currency @cart.total_price %></td>
  </tr>
</table>

<hr />
<%= form_tag checkout_path, :style => "text-align: right" do |f|
  <%= link_to "Continue Shopping", root_path %>
  or
  <%= submit_tag "Checkout" %>
<% end %>
```

13. To tie it all together, we will add routes for our `cart` controller:

```
config/routes.rb
  get "cart" => "cart#show"
  post "cart/add/:id" => "cart#add", :as => :add_to_cart
  post "cart/remove/:id" => "cart#remove", :as => :remove_from_
  cart
```

14. In our product page, we will add a button for adding items to the cart:

```
app/views/ecom/products/index.html.erb
<div class="row">
  <% @products.each do |product| %>
    <div class="col-lg-4">
      <h3><%= link_to product.name, product %></h3>
      <p><%= image_tag product.image.url %></p>
      <p><b>Price:</b> <%= product.base_price %></p>
      <p><%= product.description %></p>
      <p><%= link_to( image_tag("ecom/add-to-cart-button.png"),
        add_to_cart_path(product.id)) %></p>
    </div>
  <%end%>

app/views/ecom/products/show.html.erb
<p id="notice"><%= notice %></p>

<p>
  <h2><%= @product.name %></h2>
</p>

<p>
  <strong>Sku:&nbsp;</strong>
  <%= @product.sku %>
</p>

<p>
  <strong>Price:&nbsp;</strong>
  <%= @product.base_price %>
</p>

<br/>
<p>
  <%= image_tag @product.image.url %>
</p>

<p><%= link_to( image_tag("ecom/add-to-cart-button.png"),
  add_to_
  cart_path(@product.id)) %></p>
```

Objective complete – mini debriefing

In this task, we created a very basic `cart` function. The logic behind a `cart` function is that it should be valid throughout the user session. That way the user has the flexibility to add and remove products at will while shopping. In addition, we added two more models: `line_item` and `purchase`. While in the cart, we need to keep a track of the details of the products that are in the cart; we used the line item to do this.

We first made devise aware of the namespace of our model through our routes:

```
Ecom/config/routes.rb
App/config/routes.rb
devise_for :users, {
  :class_name => "Ecom::User",
  module: :devise
}
```

We want to check whether devise is bundled with the engine or not, so we will add it as a dependency in our `gemspec` file:

```
ecom/com.gemspec
s.add_dependency "devise"
```

In order to stick with a specific version to retain its compatibility, we can define the version of the dependency:

```
s.add_dependency "devise", "3.2.3"
```

We added a protected method to check if the session variable for `cart_id` has a value or not. If the value is not present, we will create a new object for the order, as shown in the following code:

```
ecom/controllers/com/cart_controller.rb
before_action :get_cart_value

protected

def get_cart_value
  if session[:cart_id].nil?
    @cart = Purchase.create
    session[:cart_id] = @cart.id
    @cart
  else
    @cart = Purchase.find(session[:cart_id])
  end
end
```

Furthermore, we created an add method in `cart_controller.rb`. We will persist `cart_id` in the session. The purchase or order is the collection of products that a user is purchasing. So, all those values will be associated to `session[:cart_id]`.

```
def add
  @cart.save
  session[:cart_id] = @cart.id
  product = Product.find(params[:id])
  item = LineItem.new
  item.make_items(@cart.id, product.id, product.base_price)
  @cart.recalculate_price!
  flash[:notice] = "Product Added to Cart"
  redirect_to cart_path
end
```

We also added a method to call the `line_item` model, and call this model on the item object in `cart_controller`. Mongoid's `create` method allows us to directly pass the parameters and create a record, as shown in the following code:

```
def self.make_items(purchase_id, product_id, price)
  LineItem.create(purchase_id: purchase_id, product_id: product_id, price: price)
end
```

Every time a product is added to the cart, we need to recalculate the total price of the order. We created `line_items.inject` and recursively added the product prices to calculate the total price. The `inject` method accepts an array (`line_items`) as the input. It reads the entire array element by element (`line_item`) and accepts a block (`sum`). So, the `inject` method will load the entire `line_items` array and initiate a block called `sum` with a value `0.0`. When the first `line_item` array is read, the `sum` function is encountered and the value is added to the `sum` block. When the `inject` method traverses the next `line_item` array, the value is added to the last updated value in the `sum` block, as shown in the following code:

```
ecom/model/ecom/purchase.rb

def recalculate_price!
  self.total_price = line_items.inject(0.0){|sum, line_item| sum
+= line_item.price }
  save!
end
```

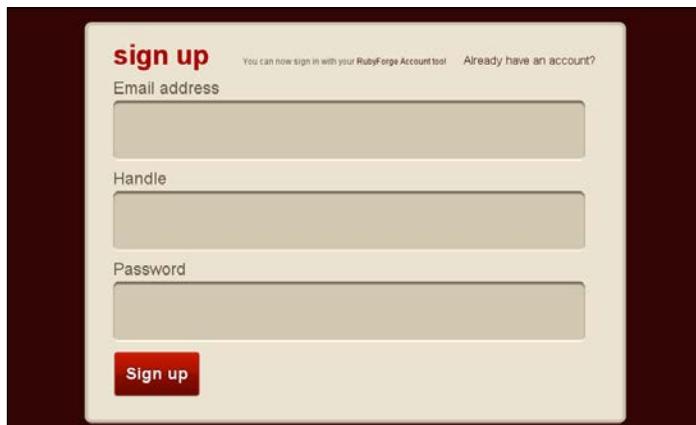
Packaging the engine as a gem

GitHub and RubyGems are the best way to host our Gems. Rubygems hosts the gem server from where people can directly install it. GitHub can be used to host the source code of the gem. We will first edit our gem and make it ready for packaging. Then, we will pack and upload it on the rubygems website.

Prepare for lift off

In order to perform this task you need to have a rubygems account and need to set it up on your local machine, as mentioned in the following steps:

1. First sign up for your account at http://rubygems.org/sign_up.



2. Then, you need to set it up on your machine through your console. Please make sure you put your handle in place of <handle> in the following code:

```
$ curl -u <handle>
https://rubygems.org/api/v1/api_key.yaml >
~/.gem/credentials; chmod 0600 ~/.gem/credentials

Enter host password for user '<handle>':
```

Engage thrusters

We will pack our newly created gem in this task:

1. In order to run the Rails engine, we need to add its base route to the `routes.rb` file of our Rails application. However, instead of asking the user to do this manually, we will create a `generators` folder inside the `lib` folder:

```
Ecom/lib$ mkdir generators  
Ecom/lib$ cd generators  
ecom/lib/generators$ mkdir ecom  
ecom/lib/generators$ mkdir templates
```

2. Inside `ecom/lib/generators`, we will create our `install` generator, as shown in the following code:

```
class Ecom::InstallGenerator < ::Rails::Generators::Base  
  include Rails::Generators::Migration  
  source_root File.expand_path('../templates', __FILE__)  
  desc "Installs Ecom Store"  
  
end
```

3. We will create an `install` method to add a line to the `routes.rb` file of our application and copy our locales to the application's `locales` folder:

```
Ecom/lib/generators/com/install_generator.rb  
  
class Ecom::InstallGenerator < ::Rails::Generators::Base  
  include Rails::Generators::Migration  
  source_root File.expand_path('../templates', __FILE__)  
  desc "Installs Ecom Store"  
  
  def install  
    route 'mount Ecom::Engine => "/store"'  
    copy_file "../../../config/locales/en.yml", "config/  
locales/com.en.yml"  
  end  
end
```

4. We will edit the `ecom.gemspec` file to add details. Make sure you add all the dependencies for the application here. Without these dependencies, the gem will not work.

```
$:.push File.expand_path("../lib", __FILE__)  
  
# Maintain your gem's version:  
require "ecom/version"
```

```
# Describe your gem and declare its dependencies:  
Gem::Specification.new do |s|  
    s.name          = "ecom"  
    s.version       = Ecom::VERSION  
    s.authors       = ["Saurabh Bhatia"]  
    s.email         = ["saurabh.a.bhatia@gmail.com"]  
    s.homepage      = "http://fedible.org"  
    s.summary        = "A Complete Ecommerce Application"  
    s.description    = "A Rails plugin to create an Ecommerce  
Application"  
  
    s.files = Dir["{app,config,db,lib}/**/*", "MIT-LICENSE",  
"Rakefile", "README.rdoc"]  
    s.test_files = Dir["test/**/*"]  
  
    s.add_dependency "rails", "~> 4.1.0.rc1"  
    s.add_dependency "mongoid", "4.0.0.beta1"  
    s.add_dependency "mongoid_slug", "3.2"  
    s.add_dependency "carrierwave", "0.10.0"  
    s.add_dependency "devise"  
end
```

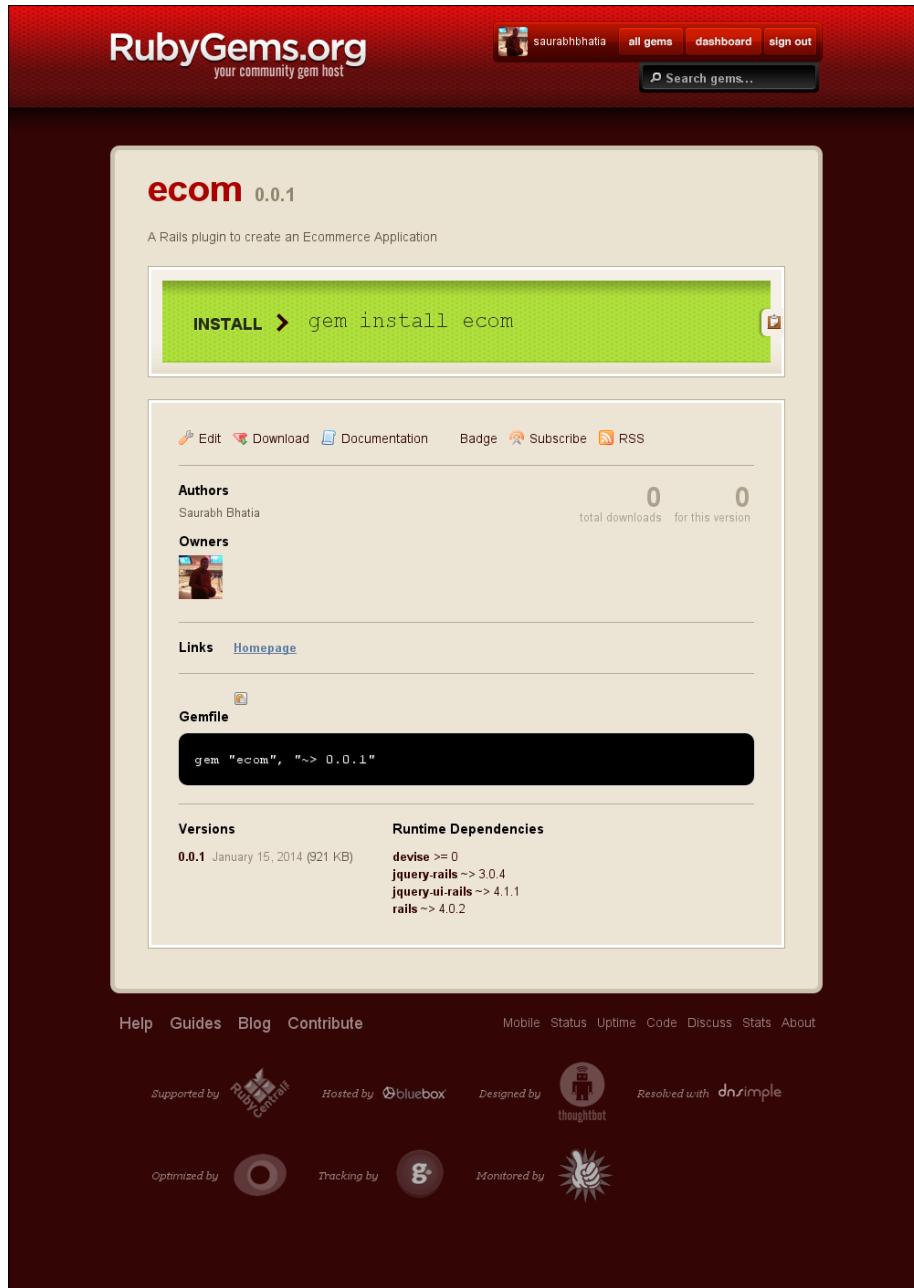
5. After the `gemspec` file is defined clearly, build the gem using `gemspec`:

```
ecom$ gem build ecom.gemspec  
Successfully built RubyGem  
Name: ecom  
Version: 0.0.1  
File: ecom-0.0.1.gem
```

6. In order to upload your gem on the rubygems server, you first need to sign up for it. If your rubygems account is correctly set up on your system, just push the gem:

```
$ gem push ecom-0.0.1.gem  
Pushing gem to https://rubygems.org...  
Successfully registered gem: ecom (0.0.1)
```

7. You will have a page created on the rubygems server for your gem, which is shown as follows:



8. We will now install a gem from our remote gem server:

```
$ gem install ecom -v 0.0.1
1 gem installed
Installing ri documentation for ecom-0.0.1...
Building YARD (yri) index for ecom-0.0.1...
```

Objective complete – mini debriefing

In this task, we prepared our application for show time by packaging it as a gem. We first began by creating a generator in our engine. This generator copies the locale files to their path and also inserts a route in the `routes.rb` file of our application. In our task, we created the generator manually. However, we can also use a generator to create a generator, shown as follows:

```
ecom$ rails g generator install
      create lib/generators/install
      create lib/generators/install/install_generator.rb
      create lib/generators/install/USAGE
      create lib/generators/install/templates
```

Then, we can add the description and tasks that the generator needs to perform. In a lot of Rails engines such as devise, the generator is used extensively to generate a user model, perform migrations, add routes to `routes.rb`, and copy locale files to the path. As you might have already seen, generators have a folder called `templates`. This folder contains templates of files that need to be copied to a particular path. For example, we need to generate a model. The generator will accept the name of the file as a command-line argument like the following code:

```
rails g model User
```

This command will copy the model for the user in the `templates` folder to the specified path and will rename it as **User**.

Rubygems has been the primary way to package and distribute Ruby programs from the beginning, be it Sinatra, only Ruby-based, or Rails engines. Rails gives us a lot of freedom to distribute a Rails engine. In case we use rubygems to distribute the engine, we will need to package the gem using the `gem build` command, as we saw in the previous task. We will then need an account on `rubygems.org` and will need to push the gem to the remote gem host. Within less than a minute, our gem is ready to be downloaded and installed. Rubygems also give some stats with the download, such as total downloads and how many downloads per day. The other way to distribute your Rails engine is directly via GitHub. If you think creating a gem is not something you want, you can host your code on GitHub and directly bundle it from there in your `Gemfile`.

The following is the screenshot of what the GitHub repository of our ecom engine looks like:

File	Description	Time Ago
app	Fixed user model	a day ago
bin	Ecommerce Plugin for Rails	5 days ago
config	Updated locales file	14 hours ago
lib	Added template for install generator	14 hours ago
test	Added Locales and Generator	a day ago
.gitignore	Merge branch 'master' of https://github.com/saurabhbhaktia/ecom	5 days ago
Gemfile	Added Slug to Products	a day ago
Gemfile.lock	Added Slug to Products	a day ago
MIT-LICENSE	Ecommerce Plugin for Rails	5 days ago
README.md	Updated Readme	4 days ago
README.rdoc	Ecommerce Plugin for Rails	5 days ago
Rakefile	Ecommerce Plugin for Rails	5 days ago
ecom.gemspec	Removed Bootstrap as gem and added it directly to asset pipeline	2 days ago

In the `Gemfile`, we will need to add something like the following code:

```
gem 'ecom', github: 'saurabhbhaktia/ecom'
```

We can also bundle a specific version, branch, tag, or a commit as follows:

```
gem 'ecom', '0.0.1', github: 'saurabhbhaktia/ecom'  
gem 'ecom', github: 'saurabhbhaktia/ecom', :branch => 'rails4'  
gem 'ecom', github: 'saurabhbhaktia/ecom', :tag => '0.0.1rc2'  
gem 'ecom', github: 'saurabhbhaktia/ecom', :ref => '151e0516'
```

Any part of the previous code can be used to bundle the gem directly from GitHub. However, we need to be sure that all values are correctly entered in `gemspec` so that it does not throw an invalid gem spec error during installation.

Mounting the engine on a blank Rails application

We have created a Rails engine with a product and cart function and even packaged it as a gem. Now, we need to take the engine for a test drive. In order to do this, we will mount it onto a blank Rails application. In this task, we will prepare and install the engine in a Rails application. We will then generate a blank Rails application and mount it onto the application.

Engage thrusters

In this task, we will mount and run a Rails engine in a Rails app. Once this is done, we will generate a blank Rails application called **Storezilla** and add our engine to the `Gemfile` by performing the following steps:

1. After adding our engine to the `Gemfile`, we will need to run the `bundle install`.

```
Gemfile
gem 'ecom', github: 'saurabhbhattacharya/ecom'
```

2. We will then run the generator we just created as follows:

```
$ rails g ecom:install
      route mount Ecom::Engine => "/store"
      create config/locales/ecom.en.yml
```

3. We can now open our `routes.rb` file and see the newly created entry as follows:

```
Storezilla::Application.routes.draw do
  get "home/index"
  mount Ecom::Engine => "/store"

  root 'home#index'
end
```

4. We will now run `rake routes` to check what routes have been created already, as follows:

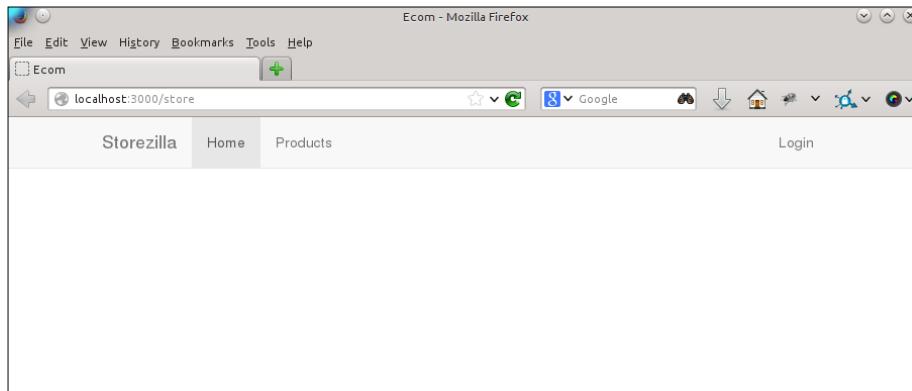
```
$ rake routes
      Prefix Verb URI Pattern          Controller#Action
home_index  GET /home/index(.:format)  home#index
      ecom      /store                 Ecom::Engine
      root     GET /                  home#index

Routes for Ecom::Engine:
      categories GET /categories(.:format)
ecom/categories#index
```

```
POST      /categories(.:format)
ecom/categories#create
      new_category GET   /categories/new(.:format)
ecom/categories#new
      edit_category GET   /categories/:id/edit(.:format)
ecom/categories#edit
      category GET   /categories/:id(.:format)
ecom/categories#show
      PATCH    /categories/:id(.:format)
ecom/categories#update
      PUT     /categories/:id(.:format)
ecom/categories#update
      DELETE   /categories/:id(.:format)
ecom/categories#destroy
      new_user_session GET   /users/sign_in(.:format)
devise/sessions#new
      user_session POST   /users/sign_in(.:format)
devise/sessions#create
      destroy_user_session DELETE /users/sign_out(.:format)
devise/sessions#destroy
      user_password POST   /users/password(.:format)
devise/passwords#create
      new_user_password GET   /users/password/new(.:format)
devise/passwords#new
      edit_user_password GET   /users/password/edit(.:format)
devise/passwords#edit
      PATCH    /users/password(.:format)
devise/passwords#update
      PUT     /users/password(.:format)
devise/passwords#update
cancel_user_registration GET   /users/cancel(.:format)
devise/registrations#cancel
      user_registration POST   /users(.:format)
devise/registrations#create
      new_user_registration GET   /users/sign_up(.:format)
devise/registrations#new
      edit_user_registration GET   /users/edit(.:format)
devise/registrations#edit
      PATCH    /users(.:format)
devise/registrations#update
      PUT     /users(.:format)
devise/registrations#update
      DELETE   /users(.:format)
devise/registrations#destroy
      products GET   /products(.:format)
ecom/products#index
      POST    /products(.:format)
ecom/products#create
```

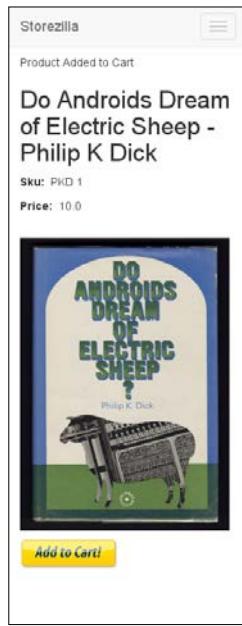
```
new_product  GET      /products/new(.:format)
ecom/products#new
    edit_product  GET      /products/:id/edit(.:format)
ecom/products#edit
    product  GET      /products/:id(.:format)
ecom/products#show
    PATCH    /products/:id(.:format)
ecom/products#update
    PUT      /products/:id(.:format)
ecom/products#update
    DELETE   /products/:id(.:format)
ecom/products#destroy
    root    GET      /
ecom/products#index
    cart    GET      /cart(.:format)
ecom/cart#show
    add_to_cart  GET      /cart/add/:id(.:format)
ecom/cart#add
    remove_from_cart  POST     /cart/remove/:id(.:format)
ecom/cart#remove
    checkout  POST     /cart/checkout(.:format)
ecom/cart#checkout
```

5. We will now navigate to the URL where we mounted our Rails engine, that is, `localhost:3000/store`. As shown in the following screenshot, we will see a blank store page:

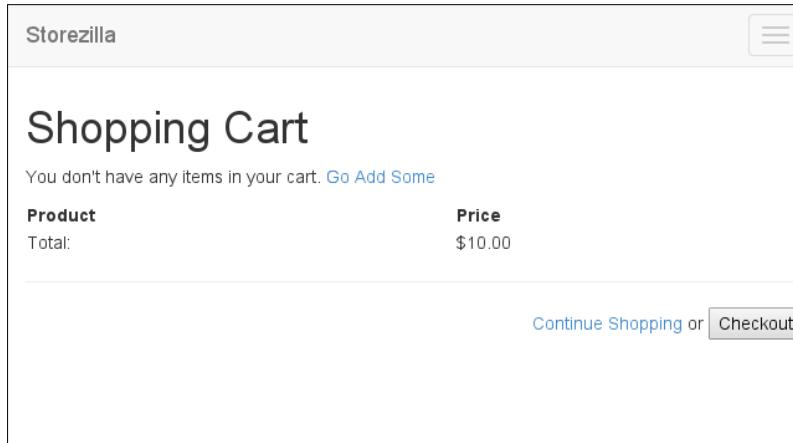


6. As we can see the store is empty, we will fill it with some products.

7. The **Products** page, after we created some products, can be browsed at <http://localhost:3000/store/products/do-androids-dream-of-electric-sheep-philip-k-dick> as shown in the following screenshot:



8. We will also see our **cart** page, which when empty looks like what is shown in the following screenshot:



Objective complete – mini debriefing

This task deals with the mounting of the engine in our application.

The first thing we did in this task was we added our engine to the `Gemfile` and `bundle install`. Then, we ran `generator` to install our `ecom` engine. This created `route` in our application where all the engine's routes will be mounted, as shown in the following code:

```
mount Ecom::Engine => "/store"
```

This route will generate all routes with the namespace `ecom` but will mount at `/store`. Also, to query the models from the Rails console within the application, we will have to prefix the namespace to the model name.

```
storezilla$ rails c
Loading development environment (Rails 4.0.2)
1.9.3-p327 :001 > ecom = Ecom::Product.new
=> #<Ecom::Product _id: 52d970207277751d36000000, name: nil,
description: nil, base_price: nil, sku: nil, _slugs: [], category_id:
nil, image: nil>
```

If users add some products to the cart and want to view the current items in it, they can browse to `/store/cart`, as shown in the following screenshot:

Product	Price	
Dan Brown - Inferno	\$10.00	Remove
Do Androids Dream of Electric Sheep - Philip K Dick	\$10.00	Remove
Animal Farm - George Orwell	\$9.00	Remove
Total:	\$29.00	

Continue Shopping or [Checkout](#)

Customizing and overriding the default classes

Until now, we have seen how to create a Rails engine, how to prepare and package it, and finally, how to mount it onto an application and use it. There are times when you need to add some custom code to the existing application. The engine code is not really seen in the folder. So, what do we do if we need to add new methods inside our engine classes? This task will deal with the customization of classes inside the engine. We will first create a state machine-based checkout system, without which our cart functionality is incomplete.

Engage thrusters

We will finally customize our methods by performing the following steps in order to add a checkout process:

1. First, we will create a namespace in the way we created our engine, as follows:

```
Storezilla/app/models$mkdir ecom  
Inside this namespace we will create our own model with the  
same name - purchase.rb .
```

2. In order to create a simple checkout process, we will need a state machine. We will use the state_machine library to add the following code:

```
Gemfile  
gem 'state_machine', github: 'pluginaweek/state_machine'
```

3. In order to use state_machine with Rails 4.1, we need to monkey patch our state_machine library. We will place this inside our config/initializers folder, as follows:

```
config/initializers/state_machine_patch.rb  
  
module StateMachine  
  
  module Integrations  
  
    module ActiveRecord  
  
      public :around_validation  
  
    end  
  
  end  
  
end
```

4. To override our model, we will use a decorator in Rails. We will first have to modify our engine to read the decorator directory:

```
lib/ecom/engine.rb  
  
module Ecom  
  
  class Engine < ::Rails::Engine  
  
    isolate_namespace Ecom  
  
    config.to_prepare do  
  
      Dir.glob(Rails.root + "app/decorators/**/*_decorator*.rb") .  
      each do |c|  
  
        require_dependency(c)
```

```
    end  
  end  
end
```

5. We will have to create an appropriate directory in our app folder, as follows:

```
$ storezilla/app~/mkdir decorators  
$ storezilla/app~/cd decorators  
$ storezilla/app/decorators~/mkdir models  
$ storezilla/app/decorators~/cd models  
$ storezilla/app/decorators/models~/mkdir ecom  
$ storezilla/app/decorators/models~/cd ecom  
$storezilla/app/decoratos/models/ecom~/touch purchase_decorator.  
rb
```

6. Once the gem is bundled, we will define states in our purchase model as follows:

```
app/decorators/model/ecom/purchase.rb  
  
state_machine :initial => :cart_in_progress do  
  event :transaction_successful do  
    transition :cart_in_progress => :order_placed  
  end  
end
```

7. We will open the Rails console and check how the state transition works with the following code:

```
1.9.3-p327 :004 > purchase = Ecom::Purchase.new  
=> #<Ecom::Purchase _id: 52dfdb787277752b8d010000, created_at:  
nil, updated_at: nil, user_id: nil, checked_out_at: nil, total_  
price: nil, state: "cart_in_progress">
```

8. We will check for state transition and see if it works as desired or not:

```
1.9.3-p327 :005 > purchase.transaction_successful!  
=> true  
1.9.3-p327 :006 > purchase  
=> #<Ecom::Purchase _id: 52dfdb787277752b8d010000, created_at:  
2014-01-22 14:56:04 UTC, updated_at: 2014-01-22 14:56:04 UTC,  
user_id: nil, checked_out_at: nil, total_price: nil, state:  
"order_placed">
```

9. We will put the state toggle inside an instance method in our model as follows:

```
app/decorators/model/ecom/purchase_decorator.rb  
def checkout!  
  self.transaction_successful!  
end
```

10. Now, we need a controller method to fire this state transition. So, we need to create a controller called cart and extend it from our existing CartController, as follows:

```
app/controllers/cart_controller.rb
class CartController < Ecom::CartController

  def checkout
    @cart.checkout!
    session.delete(:cart_id)
    flash[:notice] = "Thank you for the Order! We will e-mail you with the shipping info."
    redirect_to root_path
  end
end
```

11. We will add a custom route for the checkout method as follows:

```
post "cart/checkout" => "cart#checkout", :as => :checkout
```

12. We will try to check out and inspect the output of our checkout method as follows:

```
Started POST "/store/cart/checkout" for 127.0.0.1 at 2014-03-08 14:53:48 +0800
Processing by Ecom::CartController#checkout as HTML
Parameters: {"utf8"=>"✓", "authenticity_token"=>"MkYrMeRwtruzR5nabghV0U4mRZI/oI0pcwFsUdXWU0=", "commit"=>"Checkout"}
  MOPED: 127.0.0.1:27017 COMMAND      database=admin command={:ismaster=>1} runtime: 0.6999ms
  MOPED: 127.0.0.1:27017 QUERY       database=project10_development collection=ecom_users select
or={"$query": {"_id":>BSON::ObjectId('52d5d19c72777517c8000000')}, "$orderby":{:_id=>1}} flags={}
  limit=-1 skip=0 batch_size=nil fields=nil runtime: 0.6270ms
  MOPED: 127.0.0.1:27017 QUERY       database=project10_development collection=ecom_purchases se
lector={"_id":>BSON::ObjectId('531abe7a7277757156020000')} flags=[] limit=0 skip=0 batch_size=nil
fields=nil runtime: 0.6025ms
  MOPED: 127.0.0.1:27017 UPDATE      database=project10_development collection=ecom_purchases se
lector={"_id":>BSON::ObjectId('531abe7a7277757156020000')} update={"$set": {"state": "order_place
d", "updated_at": "2014-03-08 06:53:48 UTC"} } flags=[]
  COMMAND      database=project10_development command={:getlasterror=>1, :w=>1} runtime: 0.7999ms
Redirected to http://localhost:3000/store/
Completed 302 Found in 13ms

Started GET "/store/" for 127.0.0.1 at 2014-03-08 14:53:48 +0800
Processing by Ecom::ProductsController#index as HTML
  MOPED: 127.0.0.1:27017 QUERY      database=project10_development collection=ecom_products sel
ector={} flags=[] limit=0 skip=0 batch_size=nil fields=nil runtime: 0.7066ms
  Rendered /home/rwub/rails4-book/book/62940S_Chapter_10/ecom/app/views/ecom/products/index.html.
erb within layouts/ecom/application (7.1ms)
  MOPED: 127.0.0.1:27017 QUERY      database=project10_development collection=ecom_users select
or={"$query": {"_id":>BSON::ObjectId('52d5d19c72777517c8000000')}, "$orderby":{:_id=>1}} flags={}
  limit=-1 skip=0 batch_size=nil fields=nil runtime: 0.7254ms
Completed 200 OK in 29ms (Views: 27.3ms)
```

13. Now that we have the status of the product, we can create a simple filter to see which orders have been completed. For this, we will add a scope to our purchase model as follows:

```
app/models/ecom/purchase.rb
  scope :order_complete, -> {where(state: "order_placed")}
```

14. We will now see the result of this scope in the Rails console as follows:

```
1.9.3-p327 :001 > purchase = Ecom::Purchase.order_complete
=> #<Mongoid::Criteria
  selector: {"state"=>"order_placed"}
  options: {}
  class:   Ecom::Purchase
  embedded: false>

1.9.3-p327 :002 > purchase = Ecom::Purchase.order_complete.last
=> #<Ecom::Purchase _id: 52dfcff772777526cf000000, created_at:
2014-01-22 14:04:46 UTC, updated_at: 2014-01-22 14:04:46 UTC,
user_id: 52d5d19c72777517c8000000, total_price: 10.0, state:
"order_placed">
```

Objective complete – mini debriefing

In this task, we used a pattern in Rails called decorators. Decorators are used to extend the engine model and the controller functionality. In our case, we enhanced our model's functionality by creating a decorator for that model. One of the ways to write a decorator is using a `class_eval` function, shown as follows:

```
Ecom::Purchase.class_eval do
end
```

This function will look for the `model` class inside the `ecom` engine and decorate it with the methods in the decorator. In the decorator, we defined the methods for `state_machine`. We have seen state machines and state transition in *Project 2, Conference and Event RSVP Management*. At the time of this writing, `state_machine` is not maintained as required; hence, it is recommended that you use other similar libraries with the same purposes such as `aasm` and `workflow`.

In order to override the controller, we created a controller called `cart` in our controllers. We extended `cart_controller` in our application from our engine's `cart` controller. This will retain all the methods just as they are, and we can write more methods inside the controller. Also, if we want to override a specific method, we will define only that method (just as we defined the `checkout` in our case) and the other methods will remain intact, which is shown as follows:

```
class CartController < Ecom::CartController
end
```

Then, we created our controller method for the checkout. We called the `checkout` method in our model to toggle the state of the cart. We deleted the cart ID from the session variable once our transaction was complete after the state toggled successfully, using the following code:

```
@cart.checkout!
  session.delete(:cart_id)
Lastly, we created scope in our model. Scope in mongoid has a slightly different syntax than the scope in ActiveRecord.
scope :order_complete, -> {where(state: "order_placed")}
```

We created a scope called `order_complete`, which fetches all the purchases whose state is `order_placed`. The `where` query is written in braces, preceded by an arrow. This is a Ruby 2 project and is the updated syntax for both mongoid and ActiveRecord.

Mission accomplished

We have successfully created a Rails engine. Good work! We managed to package quite a lot of features in our shopping cart engine. In this project, we worked on the following aspects:

- ▶ We created a mountable Rails engine without ActiveRecord
- ▶ We modified the engine to work with mongoid
- ▶ We generated the models for a product and categories
- ▶ We created a shopping cart with an `add to cart` functionality
- ▶ We also added the `remove from cart` and `line items` functions
- ▶ We prepared the package for gem and uploaded it to rubygems
- ▶ We loaded the gem in the `Gemfile` and mounted the engine onto the Rails application
- ▶ We customized the controller and model to add the functions that we needed

Hotshot challenges

We created a cool shopping cart project. You can enhance it with a lot of features:

- ▶ Create an administrator area and separate the devise login based on the roles
- ▶ Add product variants as nested attributes for a product
- ▶ Create a scope filter based on the categories
- ▶ Add checkout forms and customize the user sign up form
- ▶ Add another state of cart in progress and cart failure to checkout

Index

Symbols

`:domain => :all method 142`
`:resize_to_fit method 85`

A

[About Us](#) | [Team](#) | [Careers](#) | [Work Culture](#) | [Job Openings 166](#)
`ActionController::API module 257`
`ActionController::Base class 328`
`ActionController::Renders module 151`
`add method 338`
`Add Project button 12`
`admin dashboard`
 adding 205
 creating, for clicks displaying 205, 206
 creating, for impression values displaying 205, 206
`advanced video options`
 URL 298
`analytics dashboard`
 about 187
 analyzing part 218
 building, tasks 189
 improving 218
 recording part 218
 reporting part 218
 required software 189
 tasks 189
`API application`
 challenges 283
 security-related tricks, adding to 280, 281
`API keys`
 about 273
 creating 273-279

API only application

 about 249
 checklist 251
 creating 250
 OAuth provider screen 250

APIs

 features 220
 using 247

app folder 351

application_controller.rb file 37

application

 securing, from cross-site scripting 112, 113

application login

 creating, with Twitter 221-226

application page management

 creating 160-165

arrays 166

asset caching

 implementing 182, 183

association

 about 29

 creating, between recipe and category models
 30

authentication

 adding 31

B

Balsamiq

 URL 11

bar graph

 creating, for displaying daily visit activity 210-212

Binary JSON (BSON) 191

blank Rails application

 Rails engine, mounting on 345-349

board
about 80
models, creating 80
Bootstrap
advantages 34
Bootstrap 3 39
btn-small class 37

C

cancan gem
using 134
carrierwave gem 80
carrierwave-video
about 294
URL 298
cart function 337
cart page 348
categories
adding 23
adding, steps 23-25
cuisines 23
food preferences 23
food types 23
checkout method 354
class_eval function 353
click event 240
clicks_per_article_per_day method 208
clicks per day data
line graph, creating for 207-210
click-tracking mechanism
creating 193-195
CMS
about 153
backend 153
challenges 186
features 154
frontend 153
hotshot objectives 154
page parts 153
system installation, requirements 155
CoffeeScript 90
col-lg-2 class 36
components bar 15
components option 12
components panel 13, 14
config.param_name option 90

confirm_application_owner? method 277
content
generating 177
rendering 177-182
Content Management System. *See CMS*
create method 338
create_with_omniauth 227
cross-site scripting 112, 113
CSS 182
CSV format
data, exporting to 150
cuisines category 23
current_user method 32, 272, 277

D

Dailymotion
URL 285
daily visit activity
displaying, bar graph used 210-212
dalli gem 183
database (db)
setting up 18-21
datatypes, Mongoid
Arrays 166
Embedded documents 166
Numbers 166
Regular expressions 166
decorator directory 350
decorators
using 353
demographic-based donut chart
about 213
creating 213-216
devise
about 31-34
used, for user authentication addition 31-33
doorkeeper_for method 271
DSL (domain-specific language) 134

E

each method 202
edit methods 33
embedded documents 166
embed :id parameter 267
encode method 295, 296

ensureIndex option 198

event moderation

- about 66
- accepted members 71
- adding 66-70

event page

- about 56
- mockup, creating for 44, 45

event RSVP application

- features 42
- system installation requirements 43
- tasks 43

events

- administrating 44, 45
- creating 44-46
- date formats, customizing 47-49
- RSVP, creating for 63
- search-friendly URLs, creating 49-51
- tags, adding 52-56

export_menus format

- adding 152

Export to CSV functionality

- adding 150, 151

export_to_CSV method 150

F

Facebox

- URL 105

facebook-rails gem 103

ffmpeg

- installing 298
- URL 298

ffmpeg builds

- URL 294

file uploads

- creating 79-84

find_or_create_by method 227, 231

fixtures folder 19

food preferences category 23

food-recipe websites 9

food types category 23

free trial plan

- creating 146-148

From Scratch option 12

frontend

- creating, steps 172-176
- final output 176

fulltext method 101

full-text search

- about 95
- adding 95-99

G

gem build command 343

Generic options

- URL 298

Geocoder.coordinates method 234

geocoder gem 242

getDate() function 210

GitHub

- URL 90, 276
- using 339

globally recognized avatar. *See gravatar*

Google API

- used, for friends display on Google map 237-241

Google Maps

- data passing to, with Rails 234-236
- friends displaying, with Google API 237-241

Google-Maps-for-Rails, methods

- `_each` 240
- `_extend` 240
- `_isFunction` 240
- `_isObject` 240
- `_map` 240

gravatar

- about 61
- adding, for user 61, 62

H

Haml 166

has_and_belong_to_many (HABTM) 121

helper method 235, 302

Home key 185

home page

- mockups 16
- viewing 17

home page, final system 10

html_safe tag 241

I

id attribute 304
if user_signed_in? method 33
image
 resizing 80
Image component 14
ImageMagick
 URL 80
impressionist method 198
index 101
indexing 101
index method 20
infinitely scrollable page
 creating 87-89
infinite scroll
 downloading 88
inject method 338
install method 340
instance method 200, 351
is_impressionable method 196
isolate_namespace method 328
item model
 creating 124-128

J

job
 queuing 310-316
jquery_infinitiescroll 90
JSON
 join data, sending via 264-266

K

Kaminari
 using 87
kaminari_config.rb file 89
Khan Academy
 URL 285

L

Layout Builder icon 13
line graph
 creating, for clicks per day data plotting 207-210

location-based filters
 creating 242-247

M

mailcatcher command 106
MailCatcher web console 111
mailer service
 creating 106-111
manage method 134
map function 201
map-reduce
 about 200
 writing 200-204
Meet-ups 42
memcached
 about 183, 184
 sleep 185
menu model
 creating 124-127
menu page
 with nested form 130
MiniMagick
 using 85
MockFlow
 mockups, building 17
mockingbird
 URL 11
mockups
 about 11
 creating, steps 12-16
 for homepage 16
 for recipe page 16
mockups, tools
 Balsamiq 11
 MockFlow 11
 mockingbird 11
modal box
 creating, jQuery used 102-104
 for pin resharing 105
model class 353
MongoDB
 using 187
MongoDB database
 about 190
 creating, tasks 190-192

Mongoid
about 166
datatypes 166
monthly billing
generating 146
monthly payment model
creating 146-148
morris.js method 210
Morris.Line method 210
multitenancy
adding 144, 145
defining 145
multitier plan
about 134
creating 135-138
My Events 42

N

navbar-brand class 36
nav.pagination method 88
nearbys query 246
nested form
using 129
new application
creating, steps 18-22
new video
adding 290, 291
notes
arranging, category wise 261
categories, adding to 261-263
creating 251-261
deleting 251-261
editing 251-261
numbers 166

O

OAuth2 provider
about 268
creating 268-273
Object Document Mapper (ODM) 156
Object Relationship Mapper (ORM) 156
online pinboard
creating 77
features 78
system installation requirements 79
tasks 78

P

page parts
about 168
image 171
managing 169-171
page_relationship model 164
pages
generating 177
pages option 12
paginates_per method 90
per_page method 87
pinboard 77
Pin model
index, defining 100
pins
controllers, creating 80
resharing 102-104
pins_newsletter method 107
Pinterest
feature 91
Pinterest-style grid layout
adding 91-93
Pluck 138
post
repining 78
Products page 348

R

Rails
activities 17
used, for passing Twitter data to Google Maps 234-236
Rails 4 319
Rails 4.2 upgrade 102
Rails API 250
installing 287-290
Rails engine
about 319
backbone, creating 321-326
building, tasks 320
category, creating 321
features 319, 320
mounting, on blank Rails application 345-349
packaging, as gem 339-344
product listing 321

shopping cart, creating 329
software requirements 320
type, full engine 327, 328
type, mountable engine 328
raw tag 241
recipes
 adding 26-28
 creating 26-28
recipe variable 20
redirect_to_finish_wizard method 123
Redis
 installing 310, 311
 URL 310
reduce function 201
regular expressions 166
request.location.city method 217
request.location.country method 217
resharing feature 102
restaurants model
 creating 124-128
reusable methods
 adding 144, 145
RMagick
 installing 81
 using 85
rolify 192
rolify gem 134
RSVP for events
 creating, steps 63-65
Rubygems 339

S

SaaS-baSaaS-based applications
 URL 116
SaaS-based restaurant management application
 building 116
 system installation, requirements 117
 tasks 116, 117
sanitize_sql() method 282
scaffolding 26
scopify method 134
search-friendly URLs
 creating, for events 49-51
search functionality 95
search method 98
seeds 26

send_newsletter method 110
separate admin area
 creating 155-159
serializer class 264
set method 185
shopping cart
 about 329
 checkout process, creating 329, 330
 creating 329-336
 features, adding to 354
show method 200, 304
Sidekiq 112
Sidekiq method 313
SignUp
 organizations, creating 117-123
skeleton application
 generating 287-290
slugs
 about 49
 adding 49-51
 image 52
social recipe-sharing website
 application, building 10, 11
 features 10
 required software 11
 tasks 9
software requirements 11
Solr 95
Solr query 101
SolrTM 100
sort_by(&count) method 60
state machine-based checkout system
 creating 350-353
state_machine library 350
storage_dir 85
storage rule 85
Storezilla 345
styling classes, Bootstrap
 using 34-39
subdomains
 creating 139-143

T

tag based events
 about 58, 59
 creating 57-59

tag cloud
creating, from tags 57, 58
tagged_with method 60
tags
creating 52-56
events, adding to 52-56
technical specifications. *See user stories*
text
caching 304-309
this function 204
timeoutable module 281
Tomcat 95
to_param method 304
touch method 305, 308, 309
Twitter
used, for application login creating 221-226
Twitter API
used, for Twitter data accessing 227-230
Twitter data
accessing, Twitter API used 227-230

U

update_sanitized_params method 138
uploaded videos
displaying 299-304
uploader
testing 86
upload method 86
user
gravatar, creating for 61, 62
user authentication
adding, to website, devise used 31-33
user.has_role? method 134
user roles
creating 130
permissions framework basics, adding 131-133
user's friend
latitude details, finding 232-234
longitude details, finding 232, 233
user stories 9-11

V

validations
checklist 31
video
caching 304-309
playing 299
uploading 287
video encoding 291-298
video.js
URL 299
video panel
displaying 299
videos customization
URL 298
video-streaming website 285, 286
Vimeo
URL 285
visit-tracking mechanism
about 195
creating 196-198
visual specifications. *See mockups*

W

web-analytics tools 187
whenever gem 110
where query 354
Wireframe Project screen 12
with_subdomain method 141
Wookmark
downloading 91
Wookmark grid layout
applying, to view 100
wookmark.js. jQuery plugin 93

X

Xero 151

Y

YouTube
URL 285



Thank you for buying **Rails 4 Application Development HOTSHOT**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

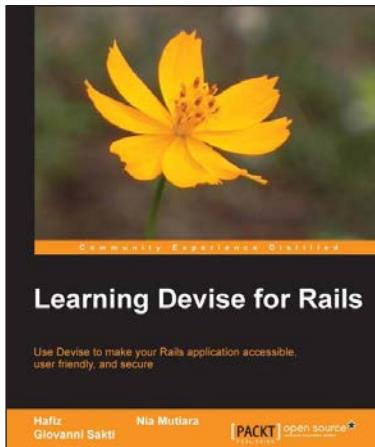
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

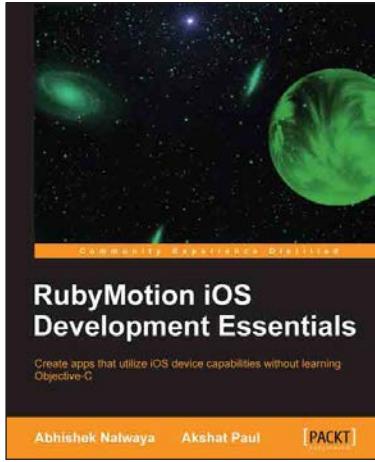


Learning Devise for Rails

ISBN: 978-1-78216-704-4 Paperback: 104 pages

Use Devise to make your Rails application accessible, user friendly, and secure

1. Use Devise to implement an e-mail-based sign-in process in a few minutes.
2. Override Devise controllers to allow username-based sign-ins, and customize default Devise HTML views to change the look and feel of the authentication system.
3. Test your authentication codes to ensure stability.



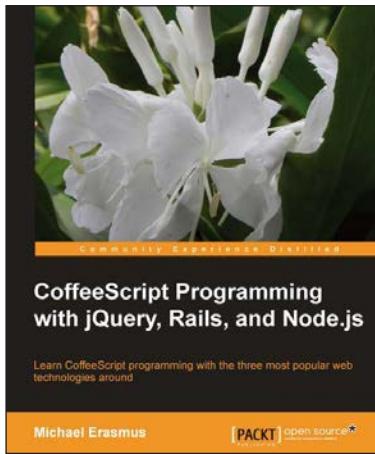
RubyMotion iOS Development Essentials

ISBN: 978-1-84969-522-0 Paperback: 262 pages

Create apps that utilize iOS device capabilities without learning Objective-C

1. Get your iOS apps ready faster with RubyMotion.
2. Use iOS device capabilities such as GPS, camera, multitouch, and many more in your apps.
3. Learn how to test your apps and launch them on the App Store.
4. Use Xcode with RubyMotion and extend your RubyMotion apps with gems.

Please check www.PacktPub.com for information on our titles

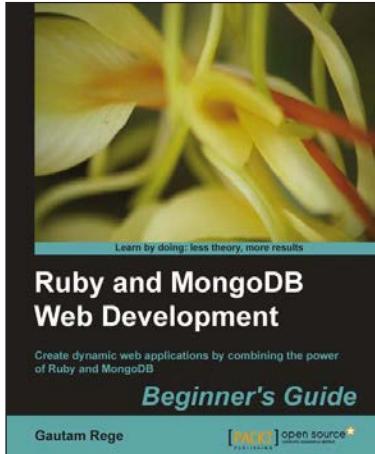


CoffeeScript Programming with jQuery, Rails, and Node.js

ISBN: 978-1-84951-958-8 Paperback: 140 pages

Learn CoffeeScript programming with the three most popular web technologies around

1. Learn CoffeeScript, a small and elegant language that compiles to JavaScript and will make your life as a web developer better.
2. Explore the syntax of the language and see how it improves and enhances JavaScript.
3. Build three example applications in CoffeeScript step by step.



Ruby and MongoDB Web Development Beginner's Guide

ISBN: 978-1-84951-502-3 Paperback: 332 pages

Create dynamic web applications by combining the power of Ruby and MongoDB

1. Step-by-step instructions and practical examples to create web applications with Ruby and MongoDB.
2. Learn to design the object model in a NoSQL way.
3. Create objects in Ruby and map them to MongoDB.

Please check www.PacktPub.com for information on our titles