TEAM RAFTAAR WEBSITE - CASE STUDY

INTRODUCTION

"TeamRataar.Com is the one stop portal for all the information regarding Team Raftaar DTU, the only HPV team that has represented India in ASME WEST."

Teamraftaar.com is an informative web application, a brochure website that displays every single bit of information about Team Raftaar DTU which is the Human Powered Vehicle Society (HPVC) of Delhi Technological University. Not only this, the visitors also have an option to donate to the team's efforts and contact the team via mailer services.

The web application uses Ruby on Rails, or Rails as a server-side framework and SQLite as the database. The front end part of the web application is done using HTML, CSS, Javascript and Bootstrap framework. It also uses Font Awesome toolkit and jQuery.

From the Ruby On Rails documentation –

Ruby on Rails, or Rails, is a server-side web application framework written in Ruby under the MIT License. Rails is a model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages. It encourages and facilitates the use of web standards such as JSON or XML for data transfer, and HTML, CSS and JavaScript for display and user interfacing. In addition to MVC, Rails emphasizes the use of other well-known software engineering patterns and paradigms, including convention over configuration (CoC), don't repeat yourself (DRY), and the active record pattern.

FRONT-END

The front end was mainly coded in HTML, CSS, Javascript and Bootstrap framework. A brief description of each is given as under:

1. **Hypertext Markup Language (HTML)**
   HTML is a markup language that web browsers use to interpret and compose text, images, and other material into visual or audible web pages. Default characteristics for every item of HTML markup are defined in the browser, and these characteristics can be altered or enhanced by the web page designer's additional use of CSS.

2. **Cascading Style Sheets (CSS)**
   CSS is a style sheet language used for describing the presentation of a document written in a markup language. Along with HTML and JavaScript, CSS is a cornerstone technology used by most websites to create visually engaging web pages, user interfaces for web applications, and user interfaces for many mobile applications.

3. **Javascript**
   JavaScript ("JS" for short) is a full-fledged dynamic programming language that, when applied to an HTML document, can provide dynamic interactivity on websites. JavaScript itself is fairly compact yet very flexible. Developers have written a large variety of tools on top of the core JavaScript language, unlocking a vast amount of extra functionality with minimum effort. These include:
   Browser Application Programming Interfaces (APIs) — APIs built into web browsers, providing functionality like dynamically creating HTML and setting CSS styles, collecting and manipulating a video stream from the user's webcam, or generating 3D graphics and audio samples.
   Third-party APIs to allow developers to incorporate functionality in their sites from other content providers, such as Twitter or Facebook.

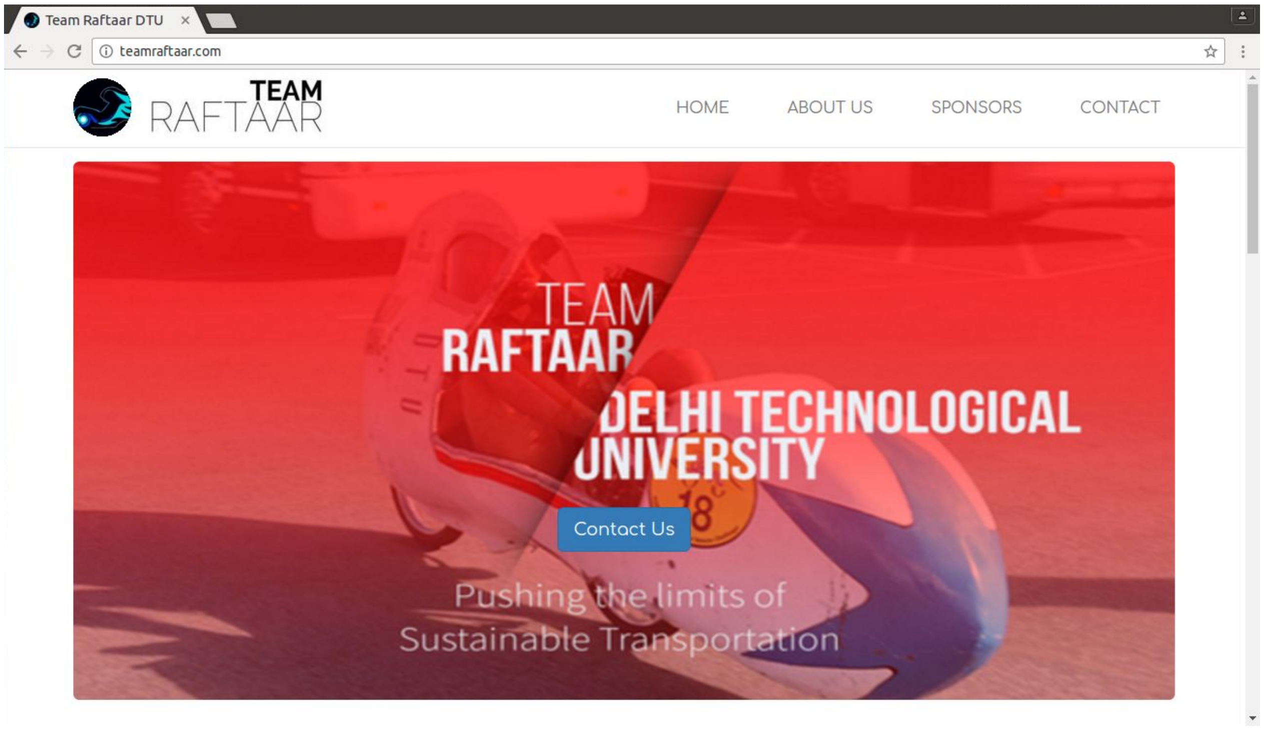
Third-party frameworks and libraries you can apply to your HTML to allow you to rapidly build up sites and applications.

4. **Bootstrap-sass v 3.3.6**
   Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery.
   bootstrap-sass is easy to drop into Rails with the asset pipeline.
   In your Gemfile you need to add the bootstrap-sass gem, and ensure that the sass-rails gem is present - it is added to new Rails applications by default.
   ```
   gem 'bootstrap-sass', '~> 3.3.6'
   gem 'sass-rails', '>= 3.2'
   ```
   bundle install and restart your server to make the files available through the pipeline.
   Import Bootstrap styles in
   `app/assets/stylesheets/application.scss:`
   ```
   // "bootstrap-sprockets" must be imported before
   "bootstrap" and "bootstrap/variables"
   @import "bootstrap-sprockets";
   @import "bootstrap";
   ```
   bootstrap-sprockets must be imported before bootstrap for the icon fonts to work.

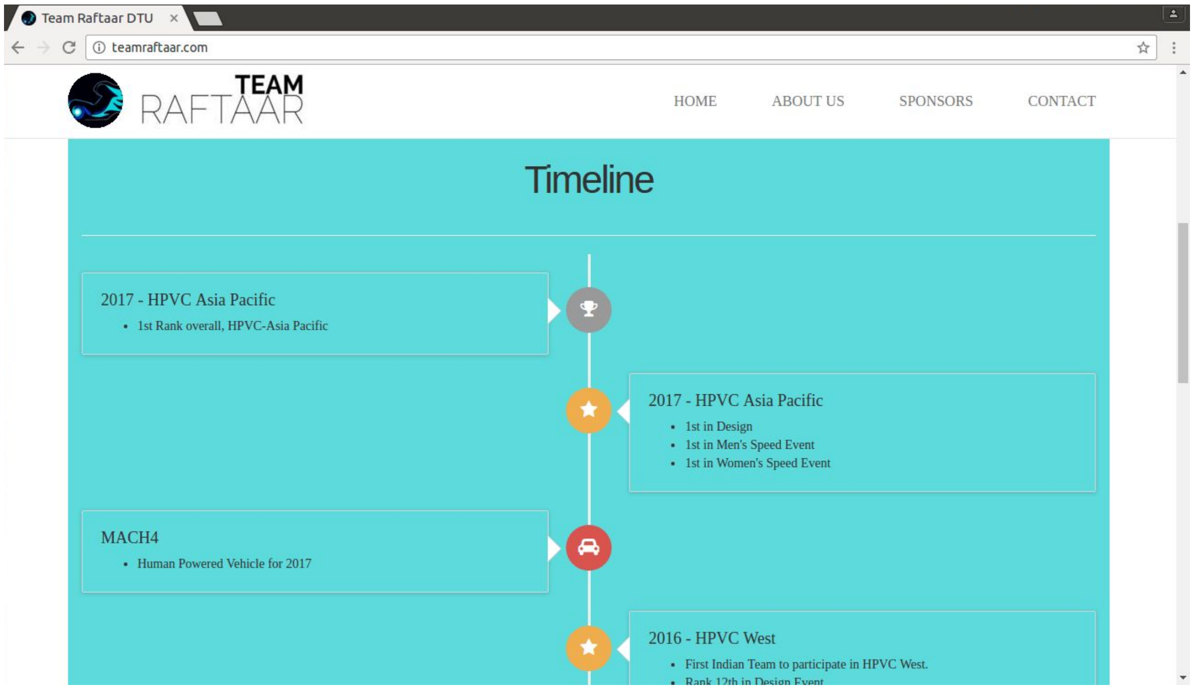
5. **JQuery-rails v 4.1.1**
   It provides jQuery and jQuery-ujs driver for the Rails 4+ application. jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

The home page () of the website if divided into four sections.

1. The first section is displayed as soon as a user open the website. It contains a captivating image, along with the team name and the college name and a slogan than best defines what the team does. It also contains a 'Contact us' button right on the center, which redirects user to the contact page. Apart from this it contains various tabs on the top right corner which allows to user to easily navigate through the various sections of the website.



On scrolling down the home page we can see a timeline that consists of all the information about the various vehicles built, the achievements of the society along with the year in which they were accomplished and some interesting facts about the vehicle built or the achievements.


Team Raftaar DTU

teamraftaar.com

TEAM RAFTAAR

HOME ABOUT US SPONSORS CONTACT

Timeline

2017 - HPVC Asia Pacific

- 1st Rank overall, HPVC-Asia Pacific

2017 - HPVC Asia Pacific

- 1st in Design
- 1st in Men's Speed Event
- 1st in Women's Speed Event

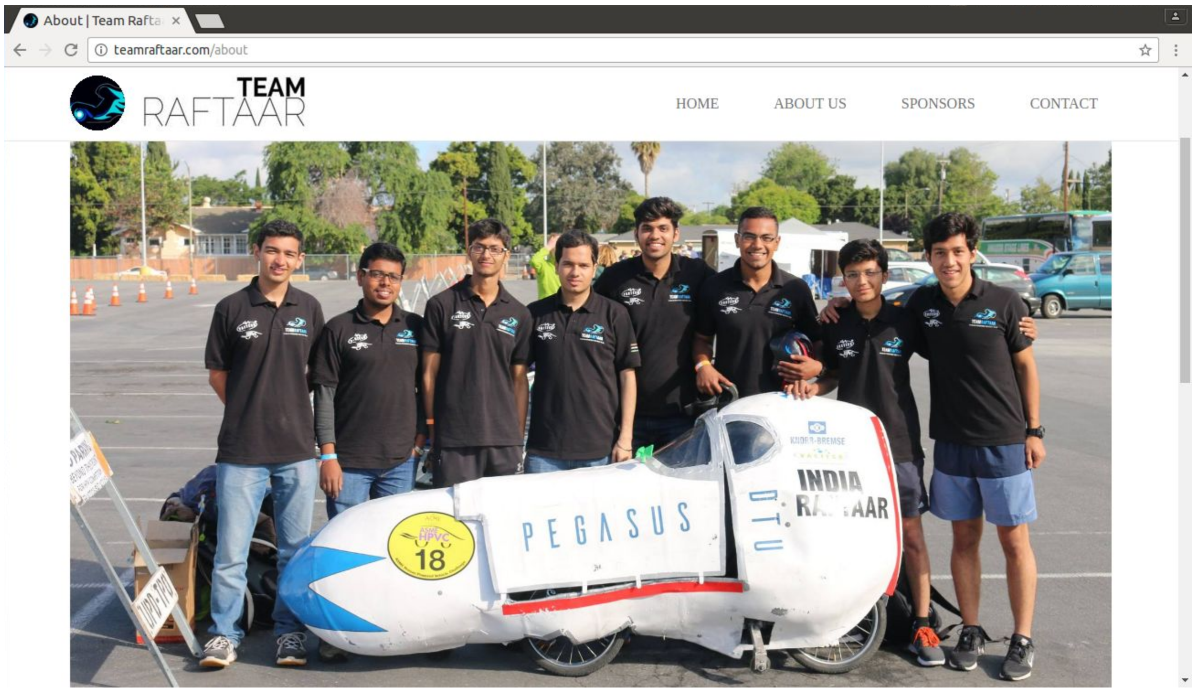
MACH4

- Human Powered Vehicle for 2017

2016 - HPVC West

- First Indian Team to participate in HPVC West.
- Rank 12th in Design Event

2. The second section is the About Us section. This section consists of information regarding the team itself, Delhi Technological University and information about ASME HPVC. It basically gives the user an idea about what the society is and what it does.

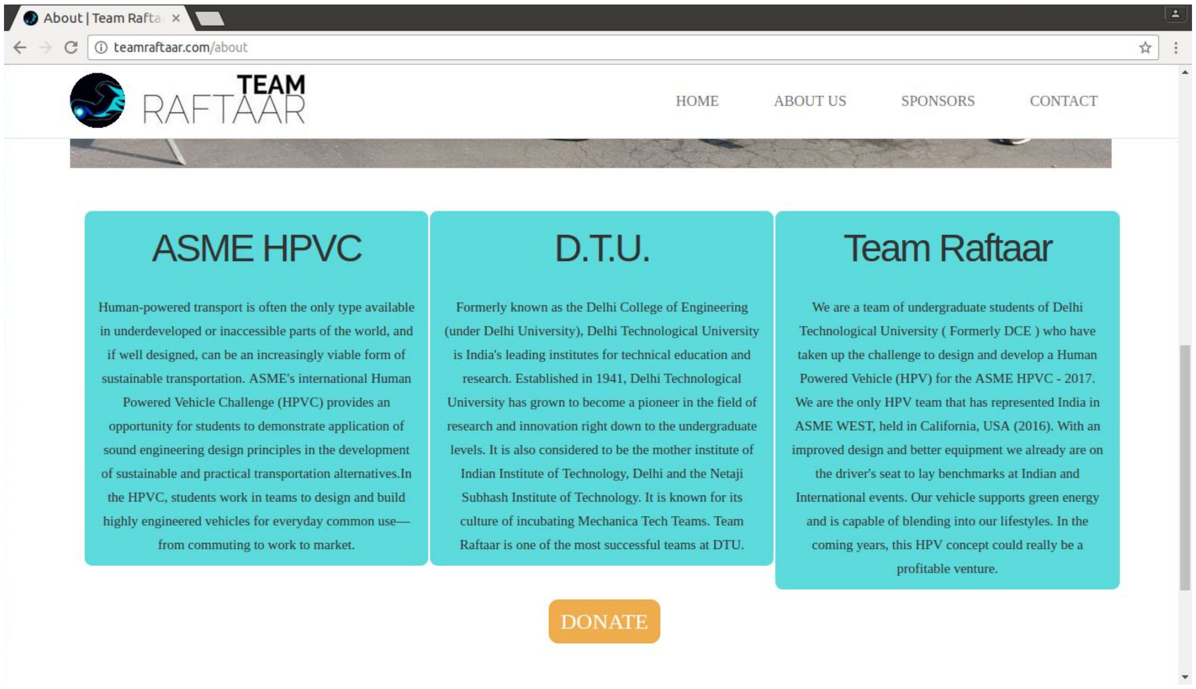

About | Team Rafta

teamraftaar.com/about

TEAM RAFTAAR

HOME ABOUT US SPONSORS CONTACT

Apart from the information the user can also use a 'Donate' button in order to donate funds according to their wish and desire to the team for their endeavours. This button redirects the user to the PayPal payment gateway of the representative of the team.



ASME HPVC

Human-powered transport is often the only type available in underdeveloped or inaccessible parts of the world, and if well designed, can be an increasingly viable form of sustainable transportation. ASME's international Human Powered Vehicle Challenge (HPVC) provides an opportunity for students to demonstrate application of sound engineering design principles in the development of sustainable and practical transportation alternatives.In the HPVC, students work in teams to design and build highly engineered vehicles for everyday common use—from commuting to work to market.

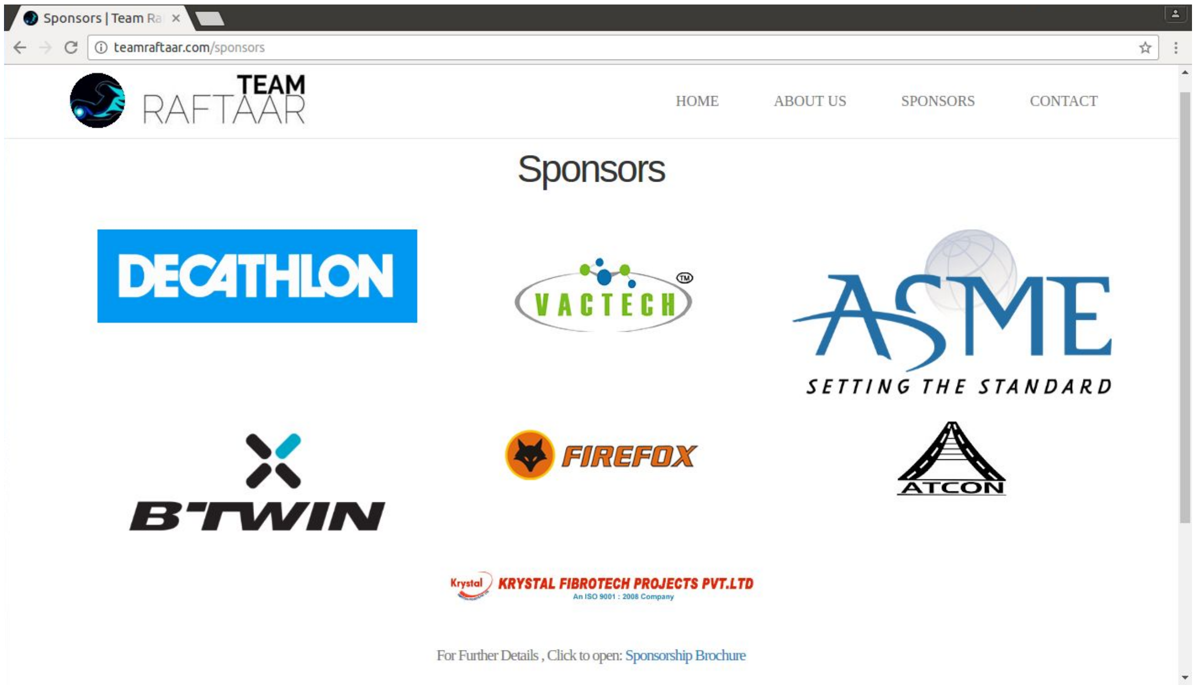
D.T.U.

Formerly known as the Delhi College of Engineering (under Delhi University), Delhi Technological University is India's leading institutes for technical education and research. Established in 1941, Delhi Technological University has grown to become a pioneer in the field of research and innovation right down to the undergraduate levels. It is also considered to be the mother institute of Indian Institute of Technology, Delhi and the Netaji Subhash Institute of Technology. It is known for its culture of incubating Mechanica Tech Teams. Team Raftaar is one of the most successful teams at DTU.

Team Raftaar

We are a team of undergraduate students of Delhi Technological University (Formerly DCE) who have taken up the challenge to design and develop a Human Powered Vehicle (HPV) for the ASME HPVC - 2017. We are the only HPV team that has represented India in ASME WEST, held in California, USA (2016). With an improved design and better equipment we already are on the driver's seat to lay benchmarks at Indian and International events. Our vehicle supports green energy and is capable of blending into our lifestyles. In the coming years, this HPV concept could really be a profitable venture.

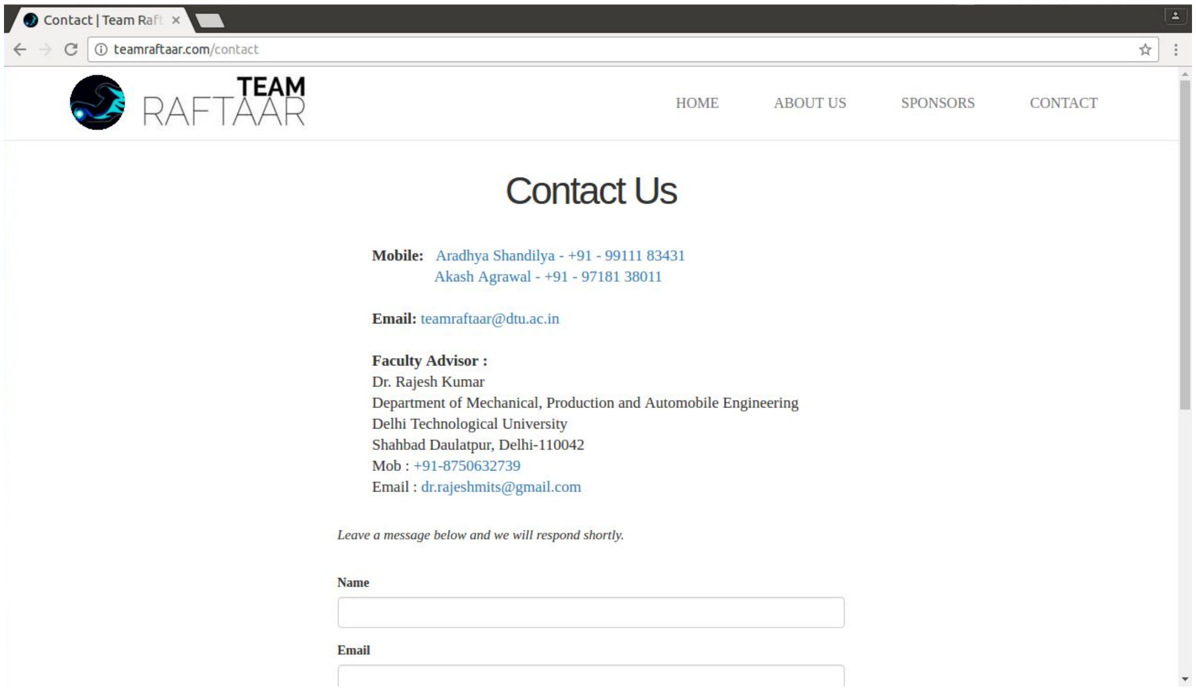
3. The third section consists of the name and mentions of the current sponsors for the team. The user can view all the current sponsors of the team and furthermore can also access the sponsorship brochure for the team, provided they are interested in knowing about how to become a sponsor for the team or the benefits and packages offered to a sponsor. This brochure consists of the complete information for sponsorship and is accessed in a pdf format by the user upon requesting access.



Sponsors

For Further Details , Click to open: Sponsorship Brochure

4. The fourth section is a contact us page that holds the contact information of the team and and also the faculty incharge of the team. Apart from this it also holds the postal address of the team incase the user needs it.



Contact Us

Mobile: Aradhya Shandilya - +91 - 99111 83431
Akash Agrawal - +91 - 97181 38011

Email: teamraftaar@dtu.ac.in

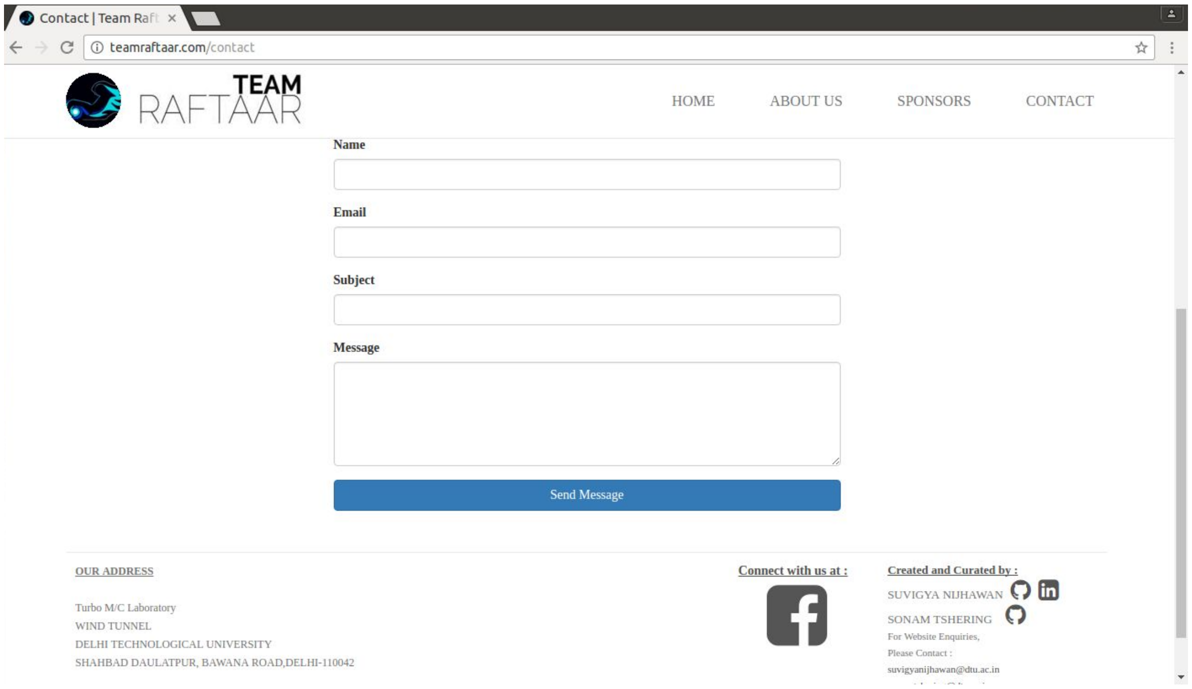
Faculty Advisor :
Dr. Rajesh Kumar
Department of Mechanical, Production and Automobile Engineering
Delhi Technological University
Shahbad Daulatpur, Delhi-110042
Mob : +91-8750632739
Email : dr.rajeshmits@gmail.com

Leave a message below and we will respond shortly.

Name

Email

There is another very crucial feature in this section i.e. the action mailer feature. Whenever the user want to send a mail to the team all the user needs to do is to fill in the details on the website itself and send the mail which automatically goes straight to the team's email inbox.



OUR ADDRESS

Turbo M/C Laboratory
WIND TUNNEL
DELHI TECHNOLOGICAL UNIVERSITY
SHAHBAD DAULATPUR, BAWANA ROAD,DELHI-110042

Connect with us at :

Created and Curated by :
SUVIGYA NIJHAWAN
SONAM TSHERING
For Website Enquiries,
Please Contact :
suvigyanijhawan@dtu.ac.in

Here is a basic guide to implementation of Action Mailers.

ACTION MAILER BASICS

1.Introduction

Action Mailer allows you to send emails from your application using mailer classes and views. Mailers work very similarly to controllers. They inherit from ActionMailer::Base and live in app/mailers, and they have associated views that appear in app/views.

2. Sending Emails

This section will provide a step-by-step guide to creating a mailer and its views.

2.1 Walkthrough to Generate a Mailer

2.1.1. Create the Mailer

```
$ bin/rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create    app/views/user_mailer
invoke  test_unit
create    test/mailers/user_mailer_test.rb
```

As you can see, you can generate mailers just like you use other generators with Rails. Mailers are conceptually similar to controllers, and so we get a mailer, a directory for views, and a test.
If you didn't want to use a generator, you could create your own file inside of app/mailers, just make sure that it inherits from `ActionMailer::Base:`

```
class MyMailer < ActionMailer::Base
end
```

2.1.2 Edit the Mailer

Mailers are very similar to Rails controllers. They also have methods called "actions" and use views to structure the content. Where a controller generates content like HTML to send back to the client, a Mailer creates a message to be delivered via email.
`app/mailers/user_mailer.rb` contains an empty mailer:

```
class UserMailer < ActionMailer::Base
  default from: 'from@example.com'
End
```

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome to My Awesome
Site')
  end
end
```

Here is a quick explanation of the items presented in the preceding method. For a full list of all available options, please have a look further down at the Complete List of Action Mailer user-settable attributes section.

- `default hash` - This is a hash of default values for any email you send from this mailer. In this case we are setting the :from header to a value for all messages in this class. This can be overridden on a per-email basis.
- `mail` - The actual email message, we are passing the :to and :subject headers in.

Just like controllers, any instance variables we define in the method become available for use in the views.

2.1.3 Create a Mailer View

Create a file called `welcome_email.html.erb` in `app/views/user_mailer/`. This will be the template used for the email, formatted in `HTML`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content='text/html; charset=UTF-8'
http-equiv='Content-Type' />
  </head>
  <body>
     <h1>Welcome to example.com, <%= @user.name %></h1>
     <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br>
    </p>
    <p>
To login to the site, just follow this link:<%= @url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

Let's also make a text part for this email. Not all clients prefer HTML emails, and so sending both is best practice. To do this, create a file called `welcome_email.text.erb` in `app/views/user_mailer/`:

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

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

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

Thanks for joining and have a great day!
```

When you call the mail method now, Action Mailer will detect the two templates (text and HTML) and automatically generate a `multipart/alternative` email.

2.1.4 Calling the Mailer

Mailers are really just another way to render a view. Instead of rendering a view and sending out the HTTP protocol, they are just sending it out through the email protocols instead. Due to this, it makes sense to just have your controller tell the Mailer to send an email when a user is successfully created.
Setting this up is painfully simple.
First, let's create a simple `User` scaffold:

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

Now that we have a user model to play with, we will just edit the `app/controllers/users_controller.rb` make it instruct the `UserMailer` to deliver an email to the newly created user by editing the create action and inserting a call to `UserMailer.welcome_email` right after the user is successfully saved:

```
class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome email after
save
        UserMailer.welcome_email(@user).deliver

        format.html { redirect_to(@user, notice: 'User was
successfully created.') }
        format.json { render json: @user, status: :created,
location: @user }
      else
        format.html { render action: 'new' }
        format.json { render json: @user.errors, status:
:unprocessable_entity }
      end
    end
  end
end
```

The method `welcome_email` returns a `Mail::Message` object which can then just be told deliver to send itself out.

2.2 Complete List of Action Mailer Methods

There are just three methods that you need to send pretty much any email message:

- `headers` - Specifies any header on the email you want. You can pass a hash of header field names and value pairs, or you can call `headers[:field_name] = 'value'`.

- `attachments` - Allows you to add attachments to your email. For example, `attachments['file-name.jpg'] = File.read('file-name.jpg').`
- `mail` - Sends the actual email itself. You can pass in headers as a hash to the mail method as a parameter, mail will then create an email, either plain text, or multipart, depending on what email templates you have defined.

2.2.1 Sending Email To Multiple Recipients

It is possible to send email to one or more recipients in one email (e.g., informing all admins of a new signup) by setting the list of emails to the `:to` key. The list of emails can be an array of email addresses or a single string with the addresses separated by commas.

```
class AdminMailer < ActionMailer::Base
  default to: Proc.new { Admin.pluck(:email) },
          from: 'notification@example.com'

  def new_registration(user)
    @user = user
    mail(subject: "New User Signup: #{@user.email}")
  end
end
```

The same format can be used to set carbon copy (Cc:) and blind carbon copy (Bcc:) recipients, by using the :cc and :bcc keys respectively.

2.2.2 Sending Email With Name

Sometimes you wish to show the name of the person instead of just their email address when they receive the email. The trick to doing that is to format the email address in the format `"Full Name <email>"`.

```
def welcome_email(user)
  @user = user
  email_with_name = "#{@user.name} <#{@user.email}>"
  mail(to: email_with_name, subject: 'Welcome to My Awesome
Site')
```

```
 end
```

2.3 Mailer Views

Mailer views are located in the `app/views/name_of_mailer_class` directory. The specific mailer view is known to the class because its name is the same as the mailer method. In our example from above, our mailer view for the `welcome_email` method will be in `app/views/user_mailer/welcome_email.html.erb` for the HTML version and `welcome_email.text.erb` for the plain text version.
To change the default mailer view for your action you do something like:

```
 class UserMailer < ActionMailer::Base
   default from: 'notifications@example.com'
   def welcome_email(user)
     @user = user
     @url  = 'http://example.com/login'
     mail(to: @user.email,
          subject: 'Welcome to My Awesome Site',
          template_path: 'notifications',
          template_name: 'another')
   end
 end
```

In this case it will look for templates at `app/views/notifications` with name another. You can also specify an array of paths for `template_path`, and they will be searched in order.
If you want more flexibility you can also pass a block and render specific templates or even render inline or text without using a template file:

```
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'
  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site') do |format|
      format.html { render 'another_template' }
      format.text { render text: 'Render text' }
    end
  end
end
```

This will render the template 'another_template.html.erb' for the HTML part and use the rendered text for the text part. The render command is the same one used inside of Action Controller, so you can use all the same options, such as `:text, :inline` etc.

2.4 Action Mailer Layouts

Just like controller views, you can also have mailer layouts. The layout name needs to be the same as your mailer, such as `user_mailer.html.erb` and `user_mailer.text.erb` to be automatically recognized by your mailer as a layout.
In order to use a different file, call layout in your mailer:

```
 class UserMailer < ActionMailer::Base
   layout 'awesome' # use awesome.(html|text).erb as the
 layout
 end
Just like with controller views, use yield to render the view
inside the layout.
You can also pass in a layout: 'layout_name' option to the
render call inside the format block to specify different layouts
for different formats:
 class UserMailer < ActionMailer::Base
   def welcome_email(user)
     mail(to: user.email) do |format|
       format.html { render layout: 'my_layout' }
       format.text
     end
   end
 end
```

Will render the HTML part using the `my_layout.html.erb` file and the text part with the usual `user_mailer.text.erb` file if it exists.

2.5 Generating URLs in Action Mailer Views

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the :host parameter yourself.
As the :host usually is consistent across the application you can configure it globally in `config/application.rb`:

```
config.action_mailer.default_url_options = { host: 'example.com'
}
```

2.5.1 generating URLs with url_for

You need to pass the `only_path: false` option when using `url_for`. This will ensure that absolute URLs are generated because the `url_for` view helper will, by default, generate relative URLs when a `:host` option isn't explicitly provided.

```
 <%= url_for(controller: 'welcome',
             action: 'greeting',
             only_path: false) %>
If you did not configure the :host option globally make sure to
pass it to url_for.
 <%= url_for(host: 'example.com',
             controller: 'welcome',
             action: 'greeting') %>
```

When you explicitly pass the `:host` Rails will always generate absolute URLs, so there is no need to pass `only_path: false`.

2.5.2 generating URLs with named routes

Email clients have no web context and so paths have no base URL to form complete web addresses. Thus, you should always use the `"_url"` variant of named route helpers.
If you did not configure the :host option globally make sure to pass it to the url helper.

```
 <%= user url(@user, host: 'example.com') %>
```

2.6 Sending Emails with Dynamic Delivery Options

If you wish to override the default delivery options (e.g. SMTP credentials) while delivering emails, you can do this using `delivery_method_options` in the mailer action.

```
class UserMailer < ActionMailer::Base
  def welcome_email(user, company)
    @user = user
    @url  = user_url(@user)
    delivery_options = { user_name: company.smtp_user,
                         password: company.smtp_password,
                         address: company.smtp_host }
    mail(to: @user.email,
         subject: "Please see the Terms and Conditions
attached",
         delivery_method_options: delivery_options)
  end
end
```

3. Receiving Emails

Receiving and parsing emails with Action Mailer can be a rather complex endeavor. Before your email reaches your Rails app, you would have had to configure your system to somehow forward emails to your app, which needs to be listening for that. So, to receive emails in your Rails app you'll need to:

- Implement a `receive` method in your mailer.
- Configure your email server to forward emails from the address(es) you would like your app to receive to `/path/to/app/bin/rails runner 'UserMailer.receive(STDIN.read)'`.

Once a method called receive is defined in any mailer, Action Mailer will parse the raw incoming email into an email object, decode it, instantiate a new mailer, and pass the email object to the mailer `receive` instance method. Here's an example:

```
class UserMailer < ActionMailer::Base
  def receive(email)
    page = Page.find_by(address: email.to.first)
    page.emails.create(
      subject: email.subject,
      body: email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          file: attachment,
          description: email.subject
        })
      end
    end
  end
end
```

4. Action Mailer Configuration

The following configuration options are best made in one of the environment files (environment.rb, production.rb, etc...)

Configuration	Description
`logger`	Generates information on the mailing run if available. Can be set to `nil` for no logging. Compatible with both Ruby's own `Logger` and `Log4r loggers`.

`smtp_settings`	Allows detailed configuration for `:smtp` delivery method: ● `:address` - Allows you to use a remote mail server. Just change it from its default "localhost" setting. ● `:port` - On the off chance that your mail server doesn't run on port 25, you can change it. ● `:domain` - If you need to specify a HELO domain, you can do it here. ● `:user_name` - If your mail server requires authentication, set the username in this setting. ● `:password` - If your mail server requires authentication, set the password in this setting. ● `:authentication` - If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of `:plain, :login, :cram_md5.` ● `:enable_starttls_auto` - Set this to `false` if there is a problem with your server certificate that you cannot resolve.
`sendmail_settings`	Allows you to override options for the :sendmail delivery method. ● `:location` - The location of the sendmail executable. Defaults to `/usr/sbin/sendmail.` ● `:arguments` - The command line arguments to be passed to sendmail. Defaults to `-i -t.`
`raise_delivery_errors`	Whether or not errors should be raised if the email fails to be delivered. This only works if the external email server is configured for immediate delivery.

`delivery_method`	Defines a delivery method. Possible values are: ● `:smtp` (default), can be configured by using `config.action_mailer.smtp_settings.` ● `:sendmail`, can be configured by using `config.action_mailer.sendmail_settings.` ● `:file`: save emails to files; can be configured by using `config.action_mailer.file_settings.` ● `:test`: save emails to `ActionMailer::Base.deliveries` array.
`perform_deliveries`	Determines whether deliveries are actually carried out when the deliver method is invoked on the Mail message. By default they are, but this can be turned off to help functional testing.
`deliveries`	Keeps an array of all the emails sent out through the Action Mailer with `delivery_method :test`. Most useful for unit and functional testing.
`default_options`	Allows you to set default values for the mail method options (`:from, :reply_to`, etc.).

4.1 Example Action Mailer Configuration

An example would be adding the following to your appropriate `config/environments/$RAILS_ENV.rb file`:

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   location: '/usr/sbin/sendmail',
#   arguments: '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from:
'no-reply@example.com'}
```

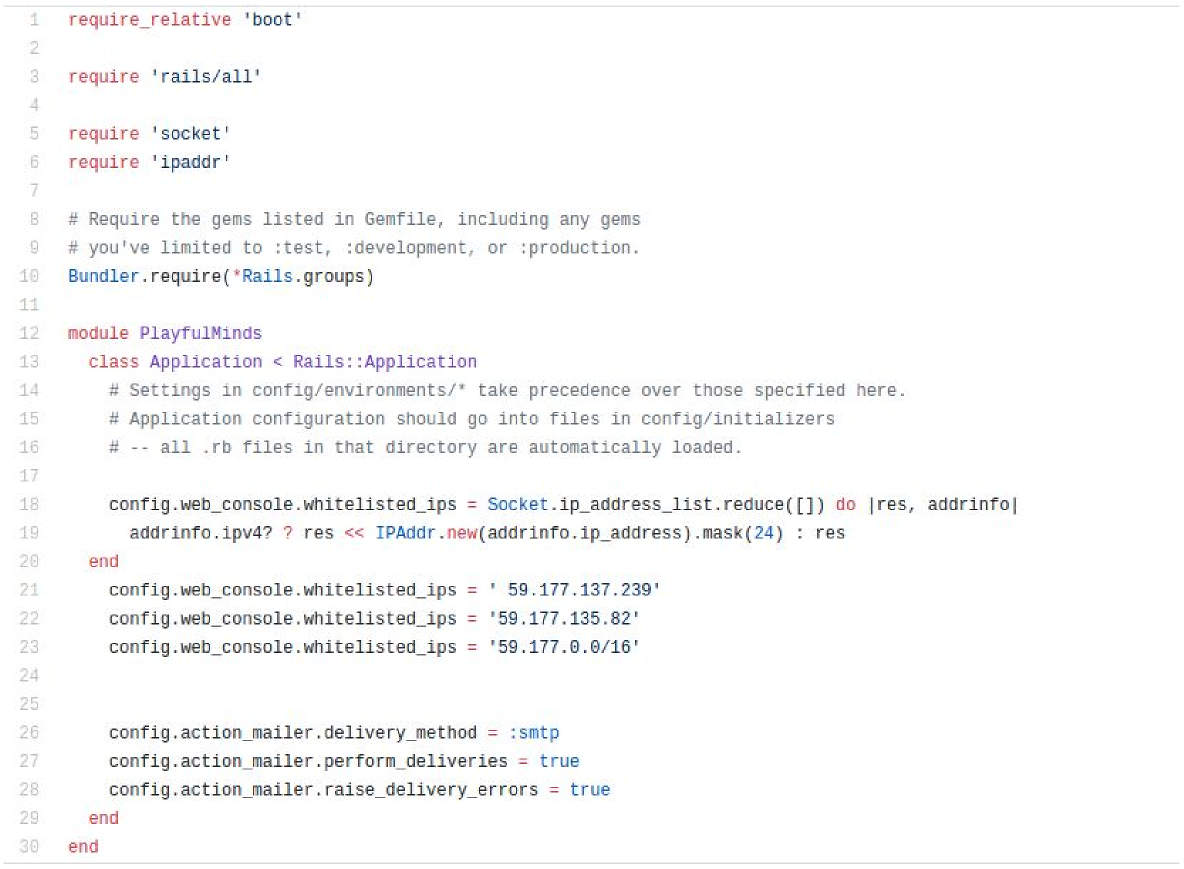
4.2 Action Mailer Configuration for Gmail

As Action Mailer now uses the Mail gem, this becomes as simple as adding to your `config/environments/$RAILS_ENV.rb` file:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:              'smtp.gmail.com',
  port:                 587,
  domain:               'example.com',
  user_name:            '<username>',
  password:             '<password>',
  authentication:       'plain',
  enable_starttls_auto: true  }
```

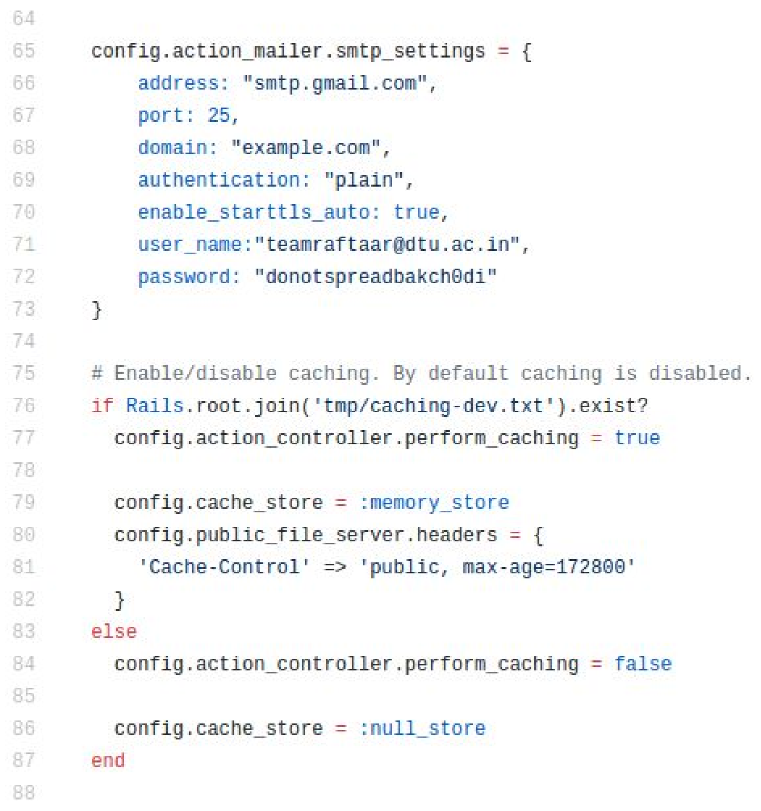
CODE SNIPPETS OF MAILER IMPLEMENTATION

Application.rb



```
require_relative 'boot'

require 'rails/all'

require 'socket'
require 'ipaddr'

# Require the gems listed in Gemfile, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(*Rails.groups)

module PlayfulMinds
  class Application < Rails::Application
    # Settings in config/environments/* take precedence over those specified here.
    # Application configuration should go into files in config/initializers
    # -- all .rb files in that directory are automatically loaded.

    config.web_console.whitelisted_ips = Socket.ip_address_list.reduce([]) do |res, addrinfo|
      addrinfo.ipv4? ? res << IPAddr.new(addrinfo.ip_address).mask(24) : res
  end
    config.web_console.whitelisted_ips = ' 59.177.137.239'
    config.web_console.whitelisted_ips = '59.177.135.82'
    config.web_console.whitelisted_ips = '59.177.0.0/16'


    config.action_mailer.delivery_method = :smtp
    config.action_mailer.perform_deliveries = true
    config.action_mailer.raise_delivery_errors = true
  end
end
```

Development.rb



```
config.action_mailer.smtp_settings = {
    address: "smtp.gmail.com",
    port: 25,
    domain: "example.com",
    authentication: "plain",
    enable_starttls_auto: true,
    user_name:"teamraftaar@dtu.ac.in",
    password: "donotspreadbakch0di"
}

# Enable/disable caching. By default caching is disabled.
if Rails.root.join('tmp/caching-dev.txt').exist?
  config.action_controller.perform_caching = true

  config.cache_store = :memory_store
  config.public_file_server.headers = {
    'Cache-Control' => 'public, max-age=172800'
  }
else
  config.action_controller.perform_caching = false

  config.cache_store = :null_store
end

```

BACK-END

The back-end part of the website was implemented using Ruby on Rails and the database was handled using SQLite.

Ruby on Rails

Rails is a web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern.
Understanding the MVC pattern is key to understanding Rails. MVC divides your application into three layers, each with a specific responsibility.
The View layer is composed of “templates” that are responsible for providing appropriate representations of your application's resources. Templates can come

in a variety of formats, but most view templates are HTML with embedded Ruby code (.erb files).
The Model layer represents your domain model (such as Account, Product, Person, Post) and encapsulates the business logic that is specific to your application. In Rails, database-backed model classes are derived from ActiveRecord::Base. Active Record allows you to present the data from database rows as objects and embellish these data objects with business logic methods. Although most Rails models are backed by a database, models can also be ordinary Ruby classes, or Ruby classes that implement a set of interfaces as provided by the ActiveModel module.
The Controller layer is responsible for handling incoming HTTP requests and providing a suitable response. Usually this means returning HTML, but Rails controllers can also generate XML, JSON, PDFs, mobile-specific views, and more. Controllers manipulate models and render view templates in order to generate the appropriate HTTP response.
In Rails, the Controller and View layers are handled together by Action Pack. These two layers are bundled in a single package due to their heavy interdependence. This is unlike the relationship between Active Record and Action Pack, which are independent. Each of these packages can be used independently outside of Rails.

Getting Started

1. Install Rails at the command prompt if you haven't yet:

```
$ gem install rails
```

2. At the command prompt, create a new Rails application:

```
$ rails new myapp
```

where “myapp” is the application name.

3. Change directory to myapp and start the web server:

```
$ cd myapp; rails server
```

Run with `--help` or `-h` for options.

4. Go to localhost:3000 and you'll see:

```
"Yay! You’re on Rails!"
```

SQLite

Stating straight from the SQLite documentation;
SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects.
SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures. These features make SQLite a popular choice as an Application File Format. Think of SQLite not as a replacement for Oracle but as a replacement for fopen()
SQLite is a compact library. With all features enabled, the library size can be less than 500 KiB, depending on the target platform and compiler optimization settings. (64-bit code is larger. And some compiler optimizations such as aggressive function inlining and loop unrolling can cause the object code to be much larger.) If optional features are omitted, the size of the SQLite library can be reduced below 300 KiB. SQLite can also be made to run in minimal stack space (4KiB) and very little heap (100 KiB), making SQLite a popular database engine choice on memory constrained gadgets such as cellphones, PDAs, and MP3 players. There is a tradeoff between memory usage and speed. SQLite generally runs faster the more memory you give it. Nevertheless, performance is usually quite good even in low-memory environments. Depending on how it is used, SQLite be faster than direct filesystem I/O.
SQLite is very carefully tested prior to every release and has a reputation for being very reliable. Most of the SQLite source code is devoted purely to testing and verification. An automated test suite runs millions and millions of test cases involving hundreds of millions of individual SQL statements and achieves 100% branch test coverage. SQLite responds gracefully to memory allocation failures

and disk I/O errors. Transactions are ACID even if interrupted by system crashes or power failures. All of this is verified by the automated tests using special test harnesses which simulate system failures. Of course, even with all this testing, there are still bugs. But unlike some similar projects (especially commercial competitors) SQLite is open and honest about all bugs and provides bugs lists and minute-by-minute chronologies of code changes.

The SQLite code base is supported by an international team of developers who work on SQLite full-time. The developers continue to expand the capabilities of SQLite and enhance its reliability and performance while maintaining backwards compatibility with the published interface spec, SQL syntax, and database file format. The source code is absolutely free to anybody who wants it, but professional support is also available.

The SQLite project was started on 2000-05-09. The future is always hard to predict, but the intent of the developers is to support SQLite through the year 2050. Design decisions are made with that objective in mind.