TAKING NASA TO THE NEXT GENERATION USING RUBY ON RAILS

A thesis written at

THE NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

and submitted to

KETTERING UNIVERSITY

in partial fulfillment

of the requirements for the

degree of

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

by

ANDREW DAVIS

September 2012

_____
Author

_____
Faculty Advisor

DISCLAIMER

This thesis is submitted as partial and final fulfillment of the cooperative work experience requirements of Kettering University needed to obtain a Bachelor of Science in Computer Engineering Degree.

The conclusions and opinions expressed in this thesis are those of the writer and do not necessarily represent the position of Kettering University or the National Aeronautical and Space Administration, or any of its directors, officers, agents, or employees with respect to the matters discussed.

PREFACE

This thesis represents the capstone of my five years combined academic work at Kettering University and job experience at the Kennedy Space Center. Academic experiences in Computer Engineering proved to be valuable assets while I developed this thesis and addressed the problem it concerns.

Although this thesis represents the compilation of my own efforts, I would like to acknowledge and extend my sincere gratitude to the following persons for their valuable time and assistance, without whom the completion of this thesis would not have been possible:

- Mr. Adam S. Dalton, Support Software Engineer at Kennedy Space Center.
- Mrs. Julie J. Peacock, Product Group Lead for Support Software at Kennedy Space Center.
- Dr. David L. Foster, Assistant Professor of Computer Engineering at Kettering University.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I. INTRODUCTION

This chapter establishes the problem topic, discusses essential background information, project criteria, methodology, and provides an overview of the supporting chapters. This thesis describes a project utilizing a new technology, Ruby on Rails, to drastically improve the contractor survey process within the Kennedy Space Center's (KSC) National Aeronautical and Space Administration (NASA) Engineering (NE) directorate and take NASA to the next generation with web applications.

Problem Topic

The workforce at KSC is largely made up of contractors. Contractors assigned to the Launch Control System (LCS) are evaluated quarterly based on a specific set of criteria. The contractor survey process requires project leads and branch chiefs to exhaust essential time. This time is largely spent performing manual calculations and copy/pasting a magnitude of data from one Excel document to another. Each survey participant not only compiles all received evaluations, but also sends their own evaluation to the supervisors who will repeat the lethargic process.

Background

In order to fully understand this thesis, a familiarity of the contractor survey process is important. As of August 2011, there were approximately 150 contractors working on the LCS project and each was assigned to a task order and report to a NASA project lead. Evaluations are performed for each task order. All of the evaluation's comments and scores are recorded in an Excel spreadsheet, which contain weightings for the different NASA branches that participate in the evaluation, as well as the criteria specified for the task order.

Task Orders

Each contract has a task order associated with it. A task order has many surveys, one for each quarter of the year. Each survey has a list of criteria associated with it that may change between surveys. The product leads fill out the task order worksheets with their scores and comments. After the document is complete, it is emailed to their branch chief. A branch chief is a supervisor of a branch, and the chiefs take all received evaluations and use them to create their own compiled evaluation. The branch chiefs take all comments and scores provided to them into careful consideration when producing the combined evaluation.

Internal Review

Each task order has a Task Order Manager (TOM). The TOM meets with all evaluators of the contractor survey for an internal review meeting. During the review, the TOM reviews each spreadsheet from the branch chiefs and generate an average score and a collective comment that represents each branch, for each criterion. This time consuming process repeats for each active survey of the quarter.

Face-To-Face

After the internal review discussion, the division chief, TOM, and the branch chiefs meet with the contractors for a face-to-face meeting, to discuss the positive and negative survey results. To help the meeting run smoothly, the division chief prints a hard copy of his evaluation for use during the meeting.

Criteria

The division chief tasked the Author to develop a database to simplify the contractor survey process. The implementation method chosen was a web application using the Ruby on Rails technology. Not only did web application need

to improve the flow of the contractor survey system, but it also was used as part of a trade study to determine if Ruby on Rails is the best tool for LCS Information Architecture development. The web application needed to be developed to allow users to log in, select an individual survey, and evaluate the survey based on pre-set criteria that has been setup by an Administrator. The web application needs to automatically perform all calculations based on the users' input.

All three roles of users (product leads, branch chiefs, and task order managers) require a simple user interface, with minimal interaction. A user wants to not only view current surveys, but also the scores and comments from previous surveys. When a lead has completed their evaluations, the ability to mark their survey as complete becomes available. Selection of this option causes the application to send emails to the appropriate individuals informing them of the completed evaluation.

Branch chiefs need similar functionality as the product leads. When a chief evaluates an active survey, they need to view all evaluations of the product leads in their branch. Before the chief submits the evaluation, they need to provide a score out of 100 for the survey. To assist with the scoring process, the application needs displays an average score of all product leads in the branch.

A TOM requires the same functionality as the leads and chiefs have, but instead of viewing comments and scores for the leads, they only see evaluations from the chiefs since the branch chief evaluations are a compilation of the lead information. A TOM needs the ability to easily see those branches that have started their evaluations, those that are complete, and those that are not started.

Any user in the system, regardless of their role, needs to be able to generate a report in PDF format of all of their evaluations for any given survey (including the past). In addition, all users need a user guide to assist with the evaluation process. A

user has the capability of having multiple roles, and to be a part of multiple branches. Necessary navigational links should be provided for these instances.

Methodology

The approach of this thesis project started with researching the Ruby on Rails web application framework. This entailed going through tutorials and developing a practice application that functioned like a blog. Utilizing the information learned from multiple sources, a prototype web application was developed that contained the bare minimum specifications, and satisfied all initial requirements. This application did not have a specialized style sheet and was not running on the latest version of the Rails framework.

During the next phase of the thesis, refactoring and redesign played a major part. The entire application was rewritten using the most up-to-date version of Ruby on Rails. The new Contractor Survey also used a stylesheet to bring out many of the applications features. Applying JavaScript for the navigation bar and data tables greatly enhanced the user experience.

The final phase is ongoing; the Author intends to continue performing maintenance to the system. This includes performing routine maintenance, implementing new features, and correcting issues. A summary of the four phases is shown in Figure 1 below.

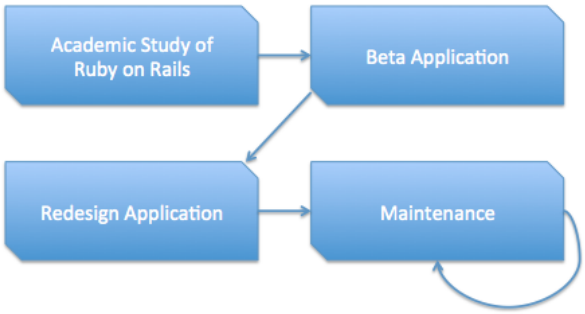

Figure 1. Methodology.

Primary Purpose

The primary purpose of this thesis is to present the impacts of the new Contractor Survey, and to describe how Ruby on Rails has helped further NASA's mission for space exploration.

Overview

The following chapters of this thesis provide conclusions about the project, as well as supporting information. The second chapter focuses on the conclusions and recommendations of the project. The third chapter discusses the technology and resources used in greater detail and how they apply to this Thesis Project. The fourth and final chapter details how Ruby on Rails is having an effect on the NASA Launch Control System Information Architecture (LCS IA) team as they migrate to this technology.

II. CONCLUSIONS AND RECOMMENDATIONS

This project has gone through many phases, all of which have been successful. The end-goal has been achieved, and a massive amount of time is being saved by allowing professionals to use the provided web application instead of a Microsoft Excel Spreadsheet.

Project Conclusions

The project as a whole has been extremely successful. A new technology, Ruby on Rails, was researched, the Author was trained to use it, and a web application was developed to simply the contractor evaluation process. The new Contractor Survey tool is currently in use in a production environment and is being maintained and upgraded on an as-needed basis. The project has exceeded the primary requirements and also meets the expectations of Kettering University for a Thesis Project.

Technology

The Author demonstrated the use of Ruby on Rails as a web application framework, and by doing so, revealed the potential of the technology for use in many other applications around the organization. Partly due to the project's success, the LCS Information Architecture (IA) team is now utilizing Ruby on Rails as the core framework of their internal application.

In addition to using a new framework during development, the Author learned how to write and execute automated testing to ensure the application is working as desired. Testing gives the developers confidence that the application is functioning properly and provides a way to show that requirements are being met. Testing can also be an interface between the stakeholders and the developers to

ensure all involved are on the same level. Testing is a vital tool designed to help developers write better web applications. Periodically, during initial development, it was necessary to deploy the Contractor Survey application to a server that others can access. To accomplish this, the Author needed to learn to use Capistrano, an unfamiliar technology to simplify the deployment process. This process is also used within LCS to deploy the IA application.

To make development more efficient, and to avoid losing previous changes, a source code control system was implemented. The Author researched how to use the source control tool to effectively manage the Contractor Survey application. Lastly, there is a vast Ruby on Rails community that is available to provide advice and to teach others about many of the prior mentioned technologies. All of the content in this section will be discussed in further detail in the next chapter.

Design

The design of the application drastically changed between the prototype version and the final product, but the usability remained the same. The Author learned about designing an application that uses proper standards for the backend to avoid messy code; this also increases the likelihood that another developer can quickly come up to speed on the project.

The initial design of the application was very basic. The idea behind it was to make a change in the way NASA LCS performs contractor evaluations, moving away from the manual labor of Excel, into a database managed web application. The first attempt at building the web application only contained the following objects to be stored in the database:

Table 1

Prototype Models

Branch	Criterion
Evaluation	ProductGroup
TaskOrder	User

The application as a whole was simple, and was used in production, but it could not meet the extra requirements that were requested for modularizing the service. When the time came to redesign the application, LCS wanted the ability to potentially share this application with other teams across the center, and maybe even the agency. In order to do so, instead of modifying the existing Contractor Survey application to meet the new requirements, it was decided that starting a new Rails application would be beneficial. The application would have the following models in its database:

Table 2

Redesigned Models

Ability	Criterion	Evaluation
Group	GroupEvaluation	GroupStatus
Membership	Status	Survey
SurveyCriterion	SurveyEvaluation	SurveyGroup
TaskOrder	TaskOrderManager	User

There are many new objects that provide the extra features throughout the entire application. The *Ability* model manages what content can be authorized by specific users who have different roles. The pre-defined *Branches* are now called *Groups*, and can be managed at a much higher level. The object *Membership* assigns *Users* into *Groups*. Both the *Status* and *GroupStatus* are used to represent

which users and groups have completed their evaluations for a given quarter. The *Evauation*, *GroupEvaluation*, and *SurveyEvaluation* all serve similar purposes, but are designed to be the three-tier hierarchy for product group leads, group representatives, and task order managers to provide their inputs.

In the new application, the TaskOrders are designed to be more modular as well. Instead of creating a new task order with the same name, but different year and quarter for each cycle, it was decided that a TaskOrder would have many surveys. This way, a survey contains the year, quarter, and criteria information (through SurveyCriterion) and the TaskOrder only contains the name, and who is serving as the TaskOrderManager.

Lessons Learned

At NASA, it is a standard in project documentation to include a section detailing many of the issues that arose, and the actions that were taken to move forward. A snippet from the internal NASA Engineering Network states, "It is important to share with others across the agency — to avoid (if a negative outcome) or support (if a positive outcome) similar situations in the future, thus ensuring that NASA learns from past successes and mistakes."

One of the most important lessons learned on this project is the need to maintain a constant form of communication between the Author and all stakeholders on the project. As development continues, it is essential to ensure that the features being implemented are really what the stakeholders want.

The Author also learned the importance of not getting discouraged easily when trying to learn a new technology, such as Ruby on Rails. Working with something unfamiliar can be a daunting task, but it is important to ask questions as they come up, instead of exhausting time that could be spent on other issues.

Future Work

As of the writing of this document, the Author believes the project is due for another redesign now that Ruby on Rails has undergone some significant changes to the framework. In order to effectively use the new technology, restarting the application from scratch would provide many benefits.

Utilizing the knowledge that the Author has gained from this Thesis Project, the application could outperform the existing application and has the potential of being more modular. There are many advanced Ruby on Rails techniques that are still being learned that, if implemented in a new version, could allow a much cleaner code repository for multiple developers to work with.

Many of the Ruby Gems (explained in the next chapter) are also frequently updated to provide a richer Application Program Interface (API) for the developers. Because of this, it is often wise to upgrade the gems and to look for alternative gems that are more useful.

III. TECHNOLOGY

This entire Thesis Project was dependent on the technology available at the time of its creation. This project would not have been possible, in a timely manner, without the use of Ruby on Rails, testing, deployment, source control, and the wonderful community to assist with development issues and provide suggestions. The following sections of this chapter will discuss each of these in detail and provide a brief understanding of how the topic relates to the thesis directly.

Rails

"Ruby on Rails is an open-source web framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration." This is a quote from the official Ruby on Rails web page. The Rails framework was developed by David Heinemier Hansson (mentioned in the Community section). Rails was extracted from a large scale commercial project, and was released as open source in 2004.

When the Author was assigned the thesis project, he needed to start from scratch. The Author did not have any prior knowledge regarding Ruby, or the web framework Rails, and needed to comprehend the following modules of the framework.

Model, View, Controller

Ruby on Rails uses the Model, View, Controller (MVC) architecture. It iss a way of organizing files, and the way in which these files interact with the client. The official Rails guides mention the following benefits:

- Isolation of business logic from the user interface
- Ease of keeping code DRY (mentioned in the next section)
- Making it clear where different types of code belong for easier maintenance

Models are basically objects, that represent data in the database. A *User* model might have attribues such as an email and encrypted password that is stored in the database. Most of the application logic belongs in the model files. *Views* act as the user interface for the application. In Ruby on Rails, views are typically written in HTML with embedded ruby inside (.html.erb). It is the job of the views to provide information to the browser that is making requests to the application. *Controllers* are the central brain of the application, they connect the models and the views. Ruby on Rails uses the controllers to processing requests from the browser, ask the appropriate models for data, and give the newly attained information to the views to be presented to the user.

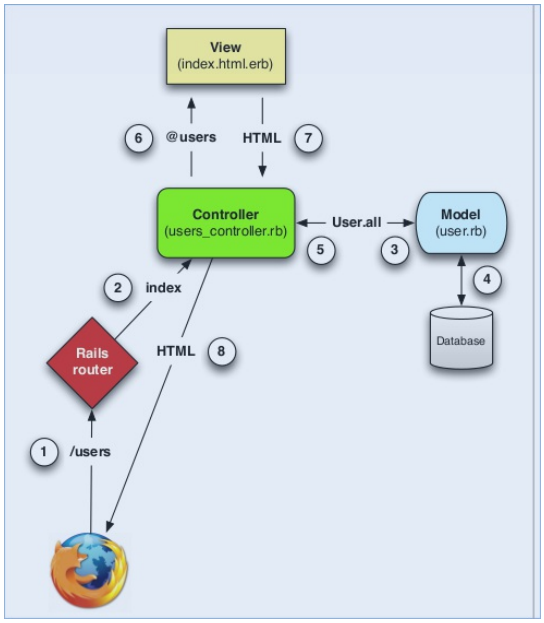

Figure 2. Model, View, Controller Diagram.

A great example of the MVC relationship is to look at Figure 2 above. In order for the Author to successfully complete this Thesis Project, a strong understanding of the MVC architecture was required. Examples of models that are

defined in the Contractor Survey application are: Criterion, Evaluation, Membership, Status, Survey, Task Order, and User. Each of these models also has a controller to manage it, and for each action in the controller, there is a view that is displayed.

For example, there is a Group model and a Group Controller that is used to Create, Read, Update, Delete (CRUD) Groups in the system. The actions in the Controller are: index, new, create, edit, update, delete. Each action is called depending on the HTTP request and the URL coming form the browser. If the user goes to https://lcscontractorsurveys.ksc.nasa.gov/groups then the index action will be called. The controller action defines a variable called *@groups* that can be used in the views. After the action is defined, a call to render the view takes place which iterates through the array of groups and displays them, as in Figure 3 below.

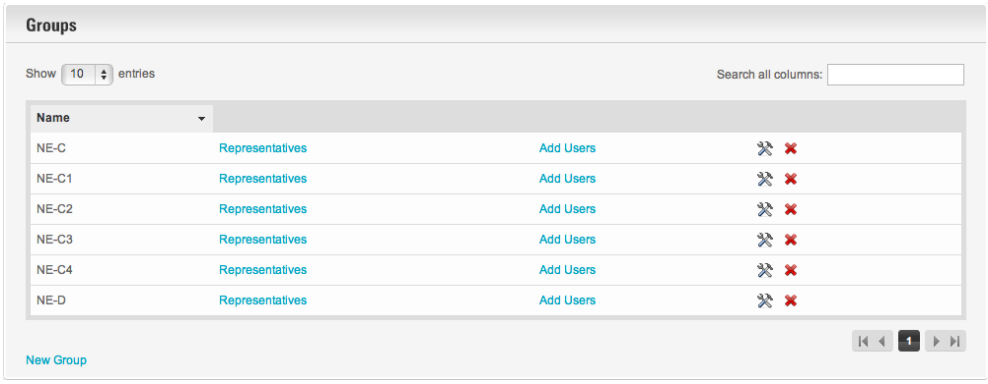

Figure 3. Group Index View.

If the user chooses to create a new group, and they click the link to do so, the URL will direct the user to */groups/new* and the controller will take control and direct the user to the *new.html.erb* view, like Figure 4.

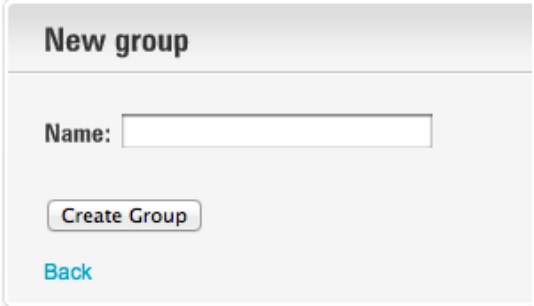

Figure 4. New Group Form.

Similar to the new action, when the user chooses to edit an existing group, the controller will go to the edit action. This will display the *edit.html.erb* view, when the user clicks to submit the form to edit the group, the update action will be called which will use an HTTP PUT request to update the database. Figure 5 is an example of the edit group page.

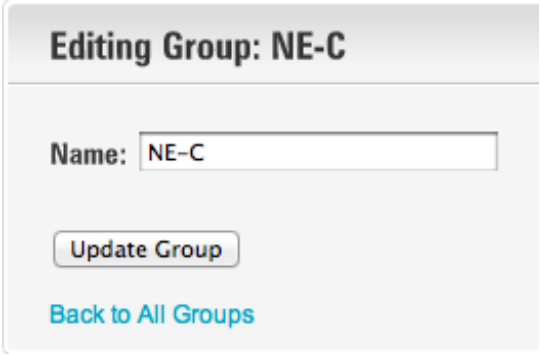

Figure 5. Edit Group Form.

Active Record

In a Ruby on Rails web application that utilizes a relational database, it is important to know the fundamentals of Active Record. Active Record is part of the Ruby on Rails framework that links ruby objects directly to database tables. The official Ruby on Rails API claims that any change in the database is instantly reflected in the Active Record objects. Since Rails is known for *convention over configuration*, the object mapping to a given Active Record class will typically happen automatically due to naming conventions. If you have a table in the database called "Evaluations", then by making a Ruby class named "Evaluation" that inherits from "ActiveRecord::Base" will be mapped.

An example of this can be shown in this thesis project, the following is the database schema for the *Evaluations* table.

```
create_table "evaluations", :force => true do |t|
    t.integer  "survey_criterion_id"
    t.float    "score"
    t.text     "comment"
    t.integer  "user_id"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.integer  "group_id"
end
```

Code 1. Evaluation Schema

An evaluation has a few attributes, such as the *score* and *comment*. There are also attributes defined for the associations (*user_id, group_id, survey_criterion_id*) which will be described in the next section. The class that pairs with this table is shown below:

```
class Evaluation < ActiveRecord::Base
  ...
end
```

Code 2. Evaluation Model

There are many custom methods inside the class that are not shown here, but because the *Evaluation* class inherits from ActiveRecord::Base, the ability to make calls to the attributes in the table are now seamlessly possible. If the comment of a particular evaluation needs to be displayed in the view, the call to *@evaluation.comment* will do the trick, with *@evaluation* being defined as an instance of the *Evaluation* model.

Associations

Active Record Associations are designed to make common operations more simple and less confusing in your code. This project contains many model objects that are related to each other, an example being a *Task Order* and a *Survey*.

```
class TaskOrder < ActiveRecord::Base
end

class Survey < ActiveRecord::Base
end
```

Code 3. TaskOrder & Survey Models without Associations

By default, there are no relationships between the two Ruby classes, they needed to be related so that a task order can have many surveys, so the Author added a *task_order_id* to the *Survey* table. A survey can belong to a task order using simple Structured Query Language (SQL) queries if the *task_order_id* was set correctly. However, this can also be more work, as it would require that identifier to be set explicitly with each creation of a *Survey* object. In this thesis project, the two classes have been changed to look like this:

```
class TaskOrder < ActiveRecord::Base
  has_many :surveys
end

class Survey < ActiveRecord::Base
  belongs_to :task_order
end
```

Code 4. TaskOrder & Survey Models with Associations

By having these two extra lines of code, Rails will do the associations automatically. When working with a single instance of a task order, *@task_order*, a call to *@task_order.surveys* will perform the correct database call to retrieve an array of all surveys that have the *task_order_id* set to the ID of the current instance. The reverse is also opposite, when calling *@survey.task_order*, Rails will find and return the *Task Order* with the proper ID.

Another example of how associations are used in this thesis project can be shown in Figure 6. It shows a form an administrator can use to create a new survey attached to a task order. Then this form is submitted, it will not only create a *Survey* object and set the correct identifiers to relate it to a *TaskOrder*, but there are other objects that will be created as well. This form lists all of the *Criteria* in the database,

and allows the administrator to select which criteria apply to this survey. For each of the selected criteria, a model called a *SurveyCriterion* is created, which simply links the *Survey* and *Criterion* models. This way each task order can have many surveys, and the criteria for the surveys can change over time by adding and removing criteria. Each of the *SurveyCriterion* objects also has a weight associated with it, this figure can also change over time as different criterion become more of a priority than others.

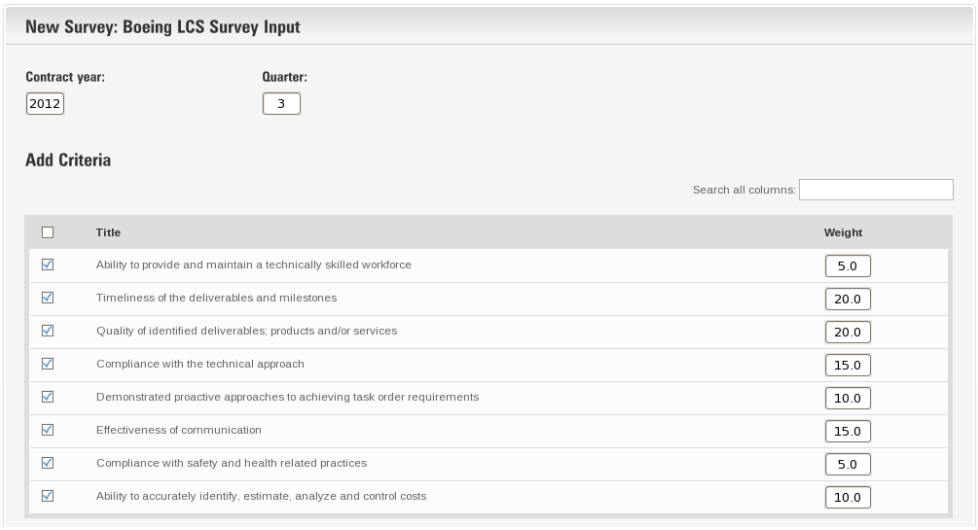

Figure 6. Rails Associations.

Active Record Associations played a major role in this thesis project. The Author needed to manage which objects had relations, and which did not. Since this thesis took an object-oriented approach, it was easier for the Author to implement as opposed to other systems.

Action Controller

Similar to Active Record, Action Controller is a fundamental part of Ruby on Rails. The Rails API claims that "Action Controllers are the core of a web request." An Action Controller is composed of a defined amount of *actions* that are executed on request, these actions will typically render a view or redirect to another action. The Rails convention is to have one controller, *ApplicationController* that

inherits from *ActionController::Base*, and all other controllers will inherit from *ApplicationController*, giving the developers the flexibility to define application-wide configurations. An example used in this project is the controller that manages the *Groups*.

```
class GroupsController < ApplicationController
  # GET /groups
  def index
    @groups = Group.all
  end
end
```

Code 5. Groups Controller: index action

In the above code, if the url of the browser points to */groups*, the router will call the *index* action on the *GroupsController*. This action makes a call to ActiveRecord to return an array of all of the *Groups* in the database and places them into an instance variable, *@groups*. Since no template is explicitly defined, Rails then automatically looks to redirect to a view located in *app/views/groups/* called *index.html.erb*. This view will be displayed to the user in the browser, and most likely will iterate through the *@groups* variable and display them all accordingly.

erb vs. haml

Ruby on Rails has two standard ways of developing the page views. The code can be written using the Embedded Ruby (.erb) format, or HTML Abstraction Markup Language (.haml). Many tutorials and training tools teach users how to use the erb format when developing, so that is what the Author decided to use during development. Both erb and haml convert into the same HTML, there is no difference in the outcome. The reason to use one over the other is for the developers to have a better understanding during development.

If the application were to be redesigned again using new technology, the Author would certainly choose to develop in haml since it is a more condensed and cleaner look to the code than erb. An example of the differences in technology is

shown in Figure 7 below.

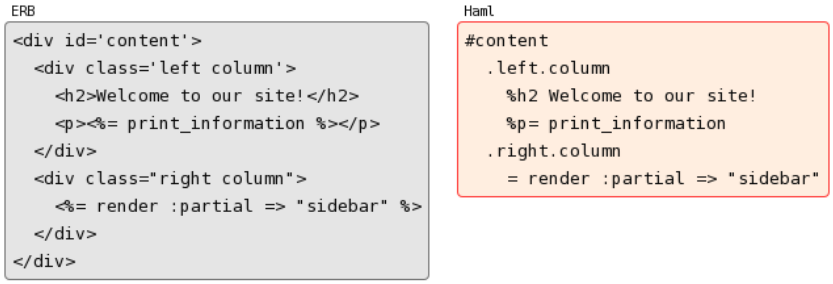

Figure 7. erb vs. haml. Note. From The Haml Team

DRY

DRY stands for "Don't Repeat Yourself". A very common phrase used in the Ruby on Rails community. DRY is the practice of not copy and pasting the same code between files. If your requirements change and ask for you to change something that is displayed on all pages, the developers would need to change each and every page individually, wasting time. With the principle of DRY, a developer can place the commonly repeated code into a single file (often called a helper or partial) and then call it from other files.

The Author needed to learn and practice this principle on a daily basis while working with the Contractor Survey. There are many pieces of code that is used across multiple pages, so it was essential to place this would-be-repeating code into a separate file, and then call it when needed. This technique is also called a "Rails Best Practice", which is a set of "guidelines" for development in Rails.

Code Reuse

When using Ruby on Rails, there is no need to *reinvent the wheel*. The Author spent a good deal of time implementing existing code into the application to make features work. When the Author needed a pagination feature, he looked to a

gem called *will_paginate*. When needing to manage all of the gem dependencies, there is a gem called *bundler* that does all of this work. Lastly, instead of writing an authentication technique from scratch, a gem named *devise* was used to handle this. More on each of these will be displayed in the next section.

Gems

The official RubyGems guides defines a gem as a software package that contains a packaged Ruby application or library. Gems are commonly used to "extend or modify functionality within a Ruby application". In fact, Ruby on Rails is a gem that provides an incredible amount of methods and libraries designed to make writing web applications more simple.

The idea of a gem is to extract methods that are not directly related to the application into a separate domain, with separate tests. When this is done, you can focus more on what the application is supposed to be doing, and errors are easier to isolate. When an error is isolated, then the potential solution can usually be found in a timely manner.

Every official gem contains the following 3 components:

- Code (including tests)
- Documentation
- gemspec

The code is the main part of the gem, that includes all of the methods that will be used, and the tests to prove that they work as desired. The documentation is designed to inform all users of the gem how it functions, any known issues, and different ways of using it. The *gemspec* is a file that details the name of the gem, the current version, author, and many more details about the gem itself.

Bundler

Bundler is a gem that is designed to make it easy to share code across a number of machines. The Bundler website describes it as a way to ensure the application has the dependencies to start up, and run without errors. Bundler will connect to all of the sources declared in the Gemfile and find all of the gems listed in the gemfile in order to meet the requirements specified. Each gem in the gemfile also has dependencies of their own, and those gems often have their own dependencies as well. Bundler is used to install the correct version numbers of all of the gems in order to satisfy the requirements.

Bundler will also inform the developer of any dependency errors. If one gem claims to need a specific version of a gem, but another gem claims to need a different version of the same gem, then there is a dependency issue that needs to be resolved. Bundler also knows if a gem that is needed is already installed on the system, and instead of downloading a new copy, it will utilize the existing copy.

When working with multiple developers, Bundler also creates a *Gemfile.lock* file that details the exact versions of all of the gems, including the third-party gems. When another developer runs *bundle install* to download and install the gems, it will refer to the Gemfile.lock to pull the same version of all third-party gems instead of checking for dependency versions again.

Devise

Devise is a RubyGem that provides a flexible authentication method that is a complete MVC solution based on Ruby on Rails engines. Devise offers 12 modules, which gives the developers the flexibility of using only the modules that are needed for the application and not worry about including more code than is being used. The Author became familiar with many of these modules, including:

- Database Authenticatable - This module encrypts and stores the password in the database as opposed to using NASA credentials.

- Recoverable - This module allows emails to be sent to the users to reset their password.

- Registerable - Instead of giving each user a username / password, this module allows users to register for the application manually. The prototype Contractor Survey used this method for the NASA personnel that needed to use the system.

- Rememberable - This module is important for keeping users happy, it allows a cookie to be created to allow users to remain logged in. If this module was not active, a user would need to log in again after each time he or she closed the browser.

- Validatable - This module ensures that the email and passwords are valid before storing them into the database.

- Lockable - In order to prevent unauthorized access, lockable will lock an account after a specified amount of failed sign-in attempts.

The above is a list of modules that were utilized throughout the Contractor Survey application in order to meet all of the project requirements. The devise gem is also very well documented and is widely-used across many Rails applications. When problems arise, many solutions have been posted on the devise wiki page to assist developers.

REST

REST stands for Representational State Transfer, it is an architecture style for web applications. A man named Roy Fielding coined the term REST in his Ph.D. disseration. REST is a stateless-architecture, "intended to evoke an image of how well-designed web applications behave", as Roy mentions in his disseration. The internet is made up of resources, an example of this relating to this Project is a Task Order. A Task Order is a resource. If clients wish to access the first Task Order in the system, they may access it with a URL like

```
http://lcscontractorsurveys.ksc.nasa.gov/task_orders/1
```

Code 6. Task Order Show URL

In this architecture, every resource has an identifier. In the case above, the identifier for the first task order is a '1'. Using a similar style as xfront.com's description of REST, when a user accesses the URL above, a representation of the resource is returned and places the client application (the browser) in a state. If the client clicks on another link, they are accessing another resource, transferring the client application to the representation of the next resource.

In order to accomplish the requirements of this thesis, the Author and his mentor decided that using Ruby on Rails would be the best approach. Ruby on Rails helps developers create web applications using the REST architecture. This allows the client to send necessary information to the server with each request, preventing the idea of having to walk through a list of steps in order to get the next representation of a resource. With REST, you can access each resource independently.

Assets

Throughout the work on this Project, the Author needed to be familiar with assets that Rails works with to function. These assets include cascading style sheets (CSS), sassy cascading style sheets (SCSS), JavaScript (JS), CoffeeScript, and images. Each of these played a significant role in making this Project successful, especially in regards to the user interface and usability of the Contractor Survey web application. The Author realized that without a good looking interface, users will be easily confused or not satisfied while using the application.

CSS

Cascading Style Sheets provide a method for styling web applications using an easy to learn syntax. Styles are defined in a .css file which inform the browser how to display HTML elements. The reason CSS exists is because HTML was never intended to contain the ability to format a document. For developers of large web pages, the document would be cluttered with font and color tags that cause confusion.

A great example of this can be shown by this document. This Thesis paper was not written using Microsoft Word, but was written in a Ruby Integrated Development Environment (IDE), then the text was compiled into HTML and CSS was applied to ensure the entire document met the formatting standards. Each indented paragraph has the following CSS applied to it:

```
p {
  color: black;
  font-family: "Times New Roman";
  font-size: 12pt;
  text-indent: 30pt;
  font-weight: normal;
  text-decoration: none;
  line-height: 2em;
}
```

Code 7. Thesis paragraph tag CSS

The Author used CSS in great abundance throughout the entire Thesis Project, ensuring things were formatted on each page of the application to meet the requirements provided. You can see from the above snippet that paragraphs like this one have a font size of 12pt, is in *Times New Roman*, and is double spaced with *line-height*. Each paragraph also has a 30pt indentation to appear like a normal indent in Microsoft Word or other word processors.

SCSS

SCSS provides a clear and concise way for writing CSS and gives the developer access to many new tools and methods. I mentioned in the previous section, CSS, that this document was written using CSS, which is technically true. However what is really happening is that the document has formatting written in SCSS to make it easier for the Author to use, then the SCSS gets compiled into CSS for the document to be created for viewing. The best way to understand SCSS is to see an example, then an explanation will follow.

```
$text-color: #000000
$font-family: "Times New Roman"
$font-size: 12pt

@mixin thesis
  color: $text-color
  font-family: $font-family
  font-size: $font-size

h1
  @include thesis
  page-break-before: always
  font-weight: bold
  text-align: center
  text-transform: uppercase

p
  @include thesis
  text-indent: 30pt
  font-weight: normal
  text-decoration: none
  line-height: 2em
```

Code 8. Thesis style using variables with SCSS

From examining the code above, the Author has set a few variables that can now be used throughout the entire document. Instead of changing the color of the headers and paragraphs separately, now the color for each HTML element is defined by a single variable. SCSS also provides something called a *@mixin*, this allows the definition of attributes to be mixed into other HTML elements. The Author uses the *thesis* mixin (shown in the code above) in both the *h1* and *p* HTML elements. SCSS follows the DRY principle mentioned earlier, allowing duplication to be eliminated

where possible.

SCSS was widely used in the creation of the Contractor Survey application to create new sections of style that the purchased stylesheet did not include, such as application specific code like the logo, etc.

JavaScript

Adding JavaScript to an application can greatly increase the usability of the application. An example of some JavaScript that is implemented in the Contractor Survey application allows supervisors to append an employee's comments to their own. Simply by clicking on a button that has the class *append_eval_btn* will call a function the Author wrote to take the comments from one user, and append it to the textbox that the supervisor is currently typing in.

```
$(document).ready(function() {
  $(".append_eval_btn").click(function() {
    var id = $(this).attr("comment_id");
    var existing = $("#evaluation_comment").val();
    var spacer = "";
    if (existing.length > 0) {
      spacer = "\n\n";
    }
    $("#evaluation_comment").val(existing + spacer + AppendText
(id, $(this)) );
  });
});
```

Code 9. JavaScript Append Function

JavaScript is useful in many web applications, such as dynamically changing the style sheets based on user actions. An example of a JavaScript feature is when the user hovers the mouse on a button, and the button changes color. JavaScript can also be used to validate forms, communicate with servers, display date pickers, etc. For the purpose of this Project, JavaScript was utilized, but was not the focal point of the project. JavaScript was only used in certain circumstances where it would increase the usability of the application as a whole.

CoffeeScript

CoffeeScript is to JavaScript as SCSS is to CSS. As mentioned before, SCSS is a simpler way for the developer to write style that is compiled into CSS. The same goes here. Developers can write CoffeeScript with ease, not having to worry about curly braces and semi-colons. Then at runtime, the CoffeeScript is compiled into JavaScript for the browser to read.

This Thesis Project does not utilize the features of CoffeeScript since it was relatively new at the time of the Project. However the Author has worked with his mentor on learning CoffeeScript and would use it for future projects. An example of the simplicity of CoffeeScript is shown below.

```
$ ->
  $(".append_eval_btn").click ->
    id = $(this).attr("comment_id")
    existing = $("#evaluation_comment").val()
    spacer = ""
    spacer = "\n\n" if existing.length > 0
    $("#evaluation_comment").val existing + spacer + AppendText
(id, $(this))
```

Code 10. Coffeescript Append Function

The snippet of CoffeeScript above compiles to be the exact JavaScript in the section above, but the CoffeeScript is written in a cleaner syntax. The CoffeeScript website says, "The golden rule of CoffeeScript is: It's just JavaScript". It is important for developers to try not to confuse the two, CoffeeScript is just a simpler way for the JavaScript to be written.

Images

In order to design a natural looking web application, it is necessary for images to be included. Many of the images were provided with the style sheet that was purchased. Others, like the NASA logo, were manually included. Images were

not a main research point in this Project, but were still necessary in order to create a user-friendly layout.

Database Authentication vs. Single Sign On

Before allowing users to use the Contractor Survey application, an authentication method needed to be decided on. there are two authentication methods that were carefully researched and considered: database authentication or single sign-on.

Database authentication can be defined as users logging into the application with a separate username and password than that of their NASA computer credentials. The devise gem was used for this method. During the prototype session of this Thesis Project, the users logged into the application using their NASA emails and passwords that they defined. Devise encrypted the passwords before storing them into the database.

Single sign-on would be ideal for allowing the NASA users to authenticate into the application without the need to remember yet another password. This process achieves a better end-result, but takes more time. In order to allow the Project to gain access to authenticating with NASA credentials, an administrator needed to be contacted to request a certificate to store in the application file structure. Figure 8 illustrates the sign-in page of the Contractor Survey using database authentication.

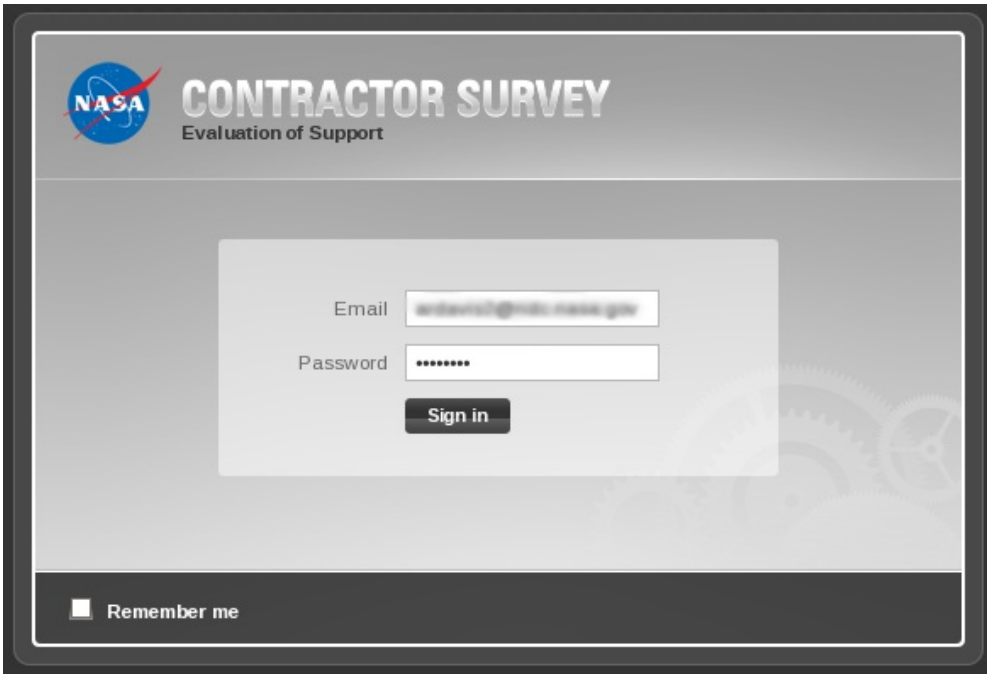

Figure 8. Database Authentication with Devise.

Testing

Ruby on Rails gives the developers multiple ways of testing web applications. It is important to test web applications during development since after making major changes, you can run your automated test suite to ensure that your application is functioning as it was before the change. It gives the developer, and the stakeholders a realization that everything is working as it should be. Two of the major types of testing that are normally involved in the development of a Ruby on Rails application are Acceptance (also called Integration) Testing and Unit Testing.

Acceptance Testing

Acceptance Tests are named as so because the tests express what the software must be able to do for the stakeholders to find it *acceptable*. This definition comes from The Cucumber Book. Cucumber is a framework designed to allow developers easily interact with the stakeholders involved on the project. The stakeholders can write actual tests that look like plain english, but are written in a specific format that can be executed. An example is shown below:

```
Feature: Creating Task Orders
  In order to create task orders
  As an admin
  I want to create, read, update, and delete Task Orders

  Background:
    Given I am logged in as an admin
    And I go to the new task order page

  Scenario: Create a task order
    When I create a task order named "Super Task Order"
    Then I should be on the "Super Task Order" task order page
    And I should be notified of the successful task order creat
ion

  Scenario: Creating a task order without a name
    When I attempt to create a task order without a name
    Then I should be informed that the task order requires a na
me
```

Code 11. Cucumber Task Order Scenarios

The code snippet above is an example of Cucumber. Cucumber will take each of those lines, and compare it against a regular expression. When it finds the appropriate step definition, shown below, code is executed that simulates the web browser and the user. This allows you to automate testing your application to ensure all of your features are working as desired.

```
When /^I create a task order named "([^"]*)"$/ do |task_order_n
ame|
  fill_in 'task_order_name', with: task_order_name
  click_button 'Create Task order'
  TaskOrder.count.should be 1
end
```

Code 12. Create Task Order Step Definition

The code above matches one of the lines in the feature using Regular Expressions, and it captures the words "Super Task Order" into the variable task_order_name. Then it will use the web browser, fill in the text box with "Super Task Order", then click the "Create Task order" button. The final line does a test to ensure that the database now has a single TaskOrder instance. If this test runs fully, the the test passes and it will move onto the next scenario or feature.

Unit Testing

The Pragmatic Programmers Book defines a *unit test* as a piece of code written by a developer that exercises a small, specific area of functionality of the code being tested. It is typical that a unit test will test a single method that the developer wrote, to ensure that it returns expected values. Without writing unit tests, your application could potentially become like a "house of cards" that is brittle and unstable. Unit tests also increase programmer happiness, since less and less time is wasted worrying about if code is working as expected.

```
describe Criterion do
  before(:each) do
    @criterion = Factory.build(:criterion)
  end

  it "should create a new instance given valid attributes" do
    Criterion.create!(@criterion.attributes)
  end

  it "should reject weight values over 100" do
    high_weight_criterion = Criterion.create!(@criterion.attrib
utes.merge(:weight => 101))
    high_weight_criterion.should_not be_valid
  end

  it "should reject weight values under 0" do
    low_weight_criterion = Criterion.create!(@criterion.attribu
tes.merge(:weight => -1))
    low_weight_criterion.should_not be_valid
  end
end
```

Code 13. Criterion Model Unit Tests

The above snippet is an example of unit tests on the Criterion model in the application. The first few lines mark a "before-each" section. This will build a criterion in memory with default values that will be used for the other tests. There are three unit tests in this section, starting with the words "it should". Unit tests are written in a descriptive context, but then Ruby code is applied to execute it. The first test validates that a Criterion can be created, given the attributes defined in the pre-built factory. The second and third tests are validating that the *weight* attribute of a Criterion cannot be above 100, or below 0.

Deployment

Deploying a Ruby on Rails application can often be difficult, but with the right documentation it can be accomplished in a decent amount of time. There is a large process involved for deploying an application, but the Author had help from the NASA Engineering hardware team to help him setup the server appropriately.

Automation

When developing the Contractor Survey, the Author needed to familiarize himself with some of the essential automation techniques related to Ruby on Rails. There are an incredible amount of command line generators that generate code to assist with development. Phusion Passenger lets developers run Ruby on Rails applications on a remote server. Capistrano assists with deploying an updated codeset to the server and tracks releases.

Generators

Throughout the development process of the Project, there is much automation taking place behind the scenes. For example, when the user wants to generate a new model for the application, there are *rake tasks* that can be run in order to create new database objects. Below is an example of creating a model called *TaskOrder* that has a title, description, and an owner.

```
rails generate model TaskOrder title:string description:text ow
ner:string
```

Code 14. Rails Generate Model Command

Running the command above will create a *migration* file, which will communicate with the database in order to create the necessary fields. An example of a migration file is shown below.

```
class CreateTaskOrders < ActiveRecord::Migration
  def self.up
    create_table :task_orders do |t|
      t.string :name
      t.timestamps
    end
  end

  def self.down
    drop_table :task_orders
  end
end
```

Code 15. Create Task Order Migration

In order to get the migrations into the database, the developer needs to run:

```
rake db:migrate
```

Code 16. Command to Migrate the Database

The command above will run through all of the migrations that have not already been run (based on the name of the file) and create appropriate tables and columns. The migration above will create a table in the database called "TaskOrders", with a "name" column. The *t.timestamps* method will also create columns such as *created_at* and *updated_at*.

Phusion Passenger

Phusion Pasenger makes the deployment of Ruby on Rails applications (and some others) simple. Passenger allows easy integration with Apache and Rails applications by allowing developers to simply upload the application source code, and not worry about configuration. David Heinemier Hansson (creator of Ruby on Rails) said, "This could become very popular, very fast!" about Phusion Passenger.

The Author assisted with the installation of Passenger onto a remote server for the Contractor Survey application to be deployed. Installing Phusion Passenger onto this server allows it be used for other Rails applications that are deployed to the same server in the future. There are other Ruby on Rails applications currently being

developed that will be deployed to the same server, those developers will not need to worry about installing Passenger.

Capistrano

Capistrano's repository on GitHub lists the product as a utility and framework for executing commands in parallel on multiple remote machines via SSH. Capistrano essentially provides a tool for developers to easily update a server with a new codebase. Capistrano is not difficult to install and includes detailed documentation for developers to get started quickly.

The Author spent a good deal of time learning about Capistrano and its capabilities. The Author's team at NASA, Information Architecture (IA), was also looking into using this technology to deploy their web application. The time spent researching and practicing with the tool was a benefit to IA as well as this Thesis Project.

One of the points the Author needed to understand was a *Capistrano Recipe*. A recipe in Capistrano is a set of tasks that can potentially be executed when the deployment command is issued. If the desired effect is to have some files copied over to a certain directory on every release, it can be done using a recipe, such as code below from the Project.

```
desc "Symlink shared configs and folders on each release."
task :symlink_shared do
  run "cp #{shared_path}/config/database.yml #{current_release}
/config/"
  run "cp #{shared_path}/Gemfile.lock #{current_release}"
  run "cp #{shared_path}/Gemfile #{current_release}/"
  run "cp #{shared_path}/.rvmrc #{current_release}/"
end
```

Code 17. Capistrano Task

The above code simply has a description of the task, the task definition, and the body of the task to copy four files from a shared directory to the current release

directory. Capistrano Recipes are great for managing how many releases the developers want to keep as well. The configuration file makes it simple to modify the location of the server to be deployed to, the username of the user, and many other options.

Source Code Control

Source Code Control (SCC) allows teams to work on the same project, sharing code amongst all members. SCC allows developers to work on local copies of files by "checking out" the file (informs others that the file is being worked on) and then "check in" the modified files when finished to a remote repository for others to use the updated software. Typically, additional features such as history, rollback, tags, branches, and locks are included. There are many source control systems such as: CVS, SVN, Git, Mercurial, and AccuRev.

Git

Git is the Author's preferred SCC. Git provides a simple command line interface for the user to manage the files of the Thesis Project, including this paper. Some sample git commands that the Author has learned and uses frequently are:

```
git pull
git checkout -b new_branch_name
git add .
git commit -m "This is a commit message"
git checkout master
git merge new_branch_name
git push
```

Code 18. Git Command Samples

These commands assist the Author with versioning the files, if there is ever code that seems unfixable, a prior secure state of code can be restored. This also protects the code from a computer failure since the code would be stored on a remote NASA server and backed up frequently.

Community

This Thesis Project would not have been possible if it were not for the Ruby on Rails community. This section is dedicated to discussing all of those individuals and organizations that played a major role in increasing the Author's knowledge on the subject, and overcoming issues. The Ruby on Rails community is increasing every day with people willing to freely support others with their expert advice, tutorials, and experiences.

Individuals

Adam Dalton

The Author was very fortunate as he had a mentor, Adam Dalton, who could sit down with him and assist in problem solving throughout all stages of the project. Adam has a vast knowledge of Ruby, and Ruby on Rails, and made it easy to learn. He had the ability of providing just enough information to make learning a challenge, which maximized the amount of material learned.

Adam took on the role of a teacher during the development of the prototype application, guiding me and showing me some of the best practices when developing. When it came time to redesign the application, Adam took on the role of a friend and colleague as we sat down together and developed the application from start to finish. Together we were able to overcome issues rather quickly and kept our ideas moving and the Project continuing forward.

Corey Haines

Corey Haines is a role model of the Rails community. His website mentions, "Corey's passion is building community, connecting people, and helping people develop their skills." Corey has inspired the Author to change his way of thinking

about problem solving, and not to be afraid of new things. He describes himself as a developer who helps developers discover ways to become better at what they do.

The Author has talked to Corey on many occasions, asking questions about different programmatic issues that arose during the development of this Thesis Project. Corey has been extremely helpful by assisting the Author tackle problems in unique ways.

Ryan Bates

Ryan Bates is the creator of Railscasts, which will be discussed in further detail later in this chapter. Ryan does a great job in helping users understand fundamental concepts of Ruby on Rails, as well as other related topics such as JavaScript, debugging, and testing. Ryan generously allowed the Author to view the Pro and Revised screencasts at no cost to learn as much as possible and proceed with the Thesis Project.

David Heinemeier Hansson

David Heinemier Hansson (DHH) is the creator of Ruby on Rails. DHH extracted the web application framework from his work on a project called Basecamp, while working at 37Signals. Since then, Rails has been released as open source, and has been continually worked on by developers around the world every day. The Author has also done his part in assisting the community by submitting a code change to the Rails source code.

The Author had an opportunity during the Thesis Project to interview DHH through an online instant messenger and gain some insight into why he created Ruby on Rails. In short, it was to increase "programmer happiness." Programmer happiness is not a phrase commonly heard, often programmers have to just deal with the stress of their work. But when using Ruby on Rails, it is open source. If there is a

problem, fix it.

Rails Tutorial

Rails Tutorial provides a free extensive Ruby on Rails tutorial. The tutorial is laid out like a book, with many chapters representing resting points. The tutorial leads the users in developing an application which behaves almost exactly like a previous version of Twitter. The book has multiple versions so it is possible to get an understanding of the content regardless of the version of Ruby on Rails the users are developing in.

CodeSchool

CodeSchool is a web application developed by EnvyLabs. This application encourages "learning by doing." Users watch a few short videos, which teach different beginner aspects about programming. In between each video, the users are challenged as they answer questions by writing code to solve simple problems. The code the user entered is then tested and appropriate feedback is returned to provide helpful hints and positive encouragement.

One of the first courses the Author took through CodeSchool is "Rails for Zombies." Rails for Zombies is a fun and interactive way of learning Ruby on Rails. It teaches you the basics and some advanced techniques to become comfortable with Ruby on Rails.

Rails Installer

When the prototype was being developed, the Author was using a Windows development machine. Often when developing on Windows, many problems arise when attempting to get the right software installed. Thankfully, RubyInstaller.org has developed a Windows executable to make the installation process simple and quick. The installer comes pre-packaged with the latest version of Ruby, Rails,

SQLite3, DevKit, and many other necessary software required to get started developing with Ruby on Rails. The Author attempted to install each of theses pieces separately at the start of the Project before being informed about Rails Installer, which made the process much more simple to get going.

Railscasts

Railscasts is a web application that is written in Ruby on Rails that has many free screencasts for viewing to learn more about using Rails. Ryan Bates devotes much of his time in recording and publishing free informational videos that teach the users how to use Rails in ways many wouldn't expect. The video is of his computer, with his voice in the background, and the users can watch and learn how a professional gets the job done.

The Author has learned many essential topics from Railscasts, especially one about upgrading an application from an older version of Ruby on Rails to a newer version without as many issues if attempting to upgrade alone. Railscasts has been a very valuable tool for learning new best practices and standards that the Ruby on Rails community has been establishing.

GitHub

GitHub is a web application, created utilizing Ruby on Rails, that allows users to host their open source and private Git repositories. Although the Author did not utilize the hosting aspects of GitHub (since NASA projects are not allowed to be hosted on a public server), there are many Ruby gems that are hosted there. Documentation for many gems are on GitHub, which provides a great interface to look up usage instructions, inform the gem creators of errors, etc.

IV. NASA ON RAILS

This chapter is dedicated to displaying how NASA is being effected by the Ruby on Rails technology. The LCS project has been significantly improved due to the framework. The following sub-sections detail how IA has increased productivity, how this Thesis Project has benefited the agency, and how Rails can continue to benefit NASA in the future.

IA

In October of 2010, a trade study was performed to determine if Ruby on Rails would benefit the IA Computer Software Configuration Item (CSCI). The trade study displays that IA was significantly behind the Build 10-1 development plan, stating that only 87 of the 402 requirements were verified. Many of the problems that have arisen from the IA development stems from the need to develop functionality in-house that was assumed to already exist or be a trivial implementation. The trade study also mentions that the program has assigned LCS IA the responsibility of a greater amount of data and the creation of more custom code to accomodate that responsibility.

The trade study reevaluated the use of the IA Semantic Web technology and associated tool sets, and the decision to be made is to continue with the current technologies or to select an alternate technology to continue with. This Thesis played a role in the decision process as it displayed that a relational database web application could be quickly developed utilizing the Ruby on Rails web framework.

Contractor Survey

Now that the Contractor Survey has been developed in Ruby on Rails, the evaluation process has significantly improved. NASA personnel spend less time dealing with frustrating Microsoft Excel documents and more time doing the tasks

assigned to them. The Kennedy Space Center is saving a substantial amount of dollars every quarter by utilizing this Project as opposed to the prior Excel system. Due to the employment of the Author as a co-op student to develop the application, the costs to produce the final product have been minimal. If this application were to be spread agency wide for similar purposes, the agency could potentially save thousands of dollars in time savings alone. However, some further maintenance would need to occur as well. The success of the Contractor Survey has served as a pilot to many other applications being developed by students at the Kennedy Space Center. They have also been designed using the Ruby on Rails framework which boosted the speed in which they were developed.

Future

NASA has a great future with Ruby on Rails. The web framework has already proved time and time again that it can assist developers with convention over configuration. It leads to programmer happiness, allowing developers to work on the meat of the applications, as opposed to spending a great deal of time configuring systems to run other technologies. Many NASA applications that are currently using Excel or physical paper are great candidates for utilizing the Ruby on Rails technology.

REFERENCES

37 Signals. Ruby on Rails Guides: Getting Started with Rails - The MVC Architecture. Retrieved April 2012, from: http://guides.rubyonrails.org/getting_started.html#the-mvc-architecture

37 Signals. Ruby on Rails Guides: A Guide to Testing Rails Applications. Retrieved April 2012, from: http://guides.rubyonrails.org/testing.html

37 Signals. Ruby on Rails. Retrieved April, 2012, from: http://http://rubyonrails.org

37 Signals. (2012, April 5). Ruby on Rails Documentation. Retrieved April 2012, from: http://api.rubyonrails.org

37 Signals. (2005, September 22). DRY In Ruby on Rails. Retrieved April 2012, from: http://oldwiki.rubyonrails.org/rails/pages/DRY

Arko, A., Balinski, M., & Bigg, R. (2012, March 02). Bundler's Purpose and Rationale. Retrieved April 2012, from: http://gembundler.com/rationale.html

Ashkenas, J. CoffeeScript. Retrieved April 2012, from: http://coffeescript.org

Catlin, H., Eppstein, C., & Weizenbaum, N. (2012). Retrieved April 2012, from: http://sass-lang.com

Costello, R. Building Web Services the REST Way. Retrieved April 2012, from: http://www.xfront.com/REST-Web-Services.html

Gunderloy, M. & Quaranto, N. What is a gem?. Retrieved April 2012, from: http://guides.rubygems.org/what-is-a-gem

Haines, C. (personal communication, 2012).

Haines, C. Corey Haines: The Software Journeyman. Retrieved April 2012, from: http://coreyhaines.com

Hambley, L., Reilly, L., & Thomas, E. (2011, October 17). Capistrano GitHub repository. Retrieved April 2012, from: https://github.com/capistrano/capistrano

Hunt, A. & Thomas, D. (2003). *Pragmatic Unit Testing: In Java with JUnit.* The Pragmatic Programmers.

NASA. (2010). *Launch Control System (LCS) Information Architecture Trade Study / Findings: Final Report.* (Adam Dalton). Kennedy Space Center, FL: Author

Phusion. Phusion Passenger. Retrieved April 2012, from: http://modrails.com

Plataformatec. (2012, May 15). Devise GitHub repository. Retrieved May 2012. from: https://github.com/plataformatec/devise

PushOk. Source Control. Retrieved April 2012, from: http://www.pushok.com/help/cvsscc/basic/why.html

The Haml Team. (2012). #haml.tutorial. Retrieved April 2012, from: http://haml.info/tutorial.html

w3schools. CSS Introduction. Retrieved April 2012, from: http://www.w3schools.com/css/css_intro.asp

w3schools. JavaScript Tutorial. Retrieved April 2012, from: http://www.w3schools.com/js

Wynne, M. & Hellesoy, A. (2012). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* The Pragmatic Programmers.

GLOSSARY

Gem	A Gem is a package of ruby code that is isolated from the rest of the application. This allows for open source code to be implemented with a simple process.
Ruby	Ruby is a scripting language, and the programming language of choice for this Project.
Ruby on Rails	Ruby on Rails is a Ruby gem that gives developers a vast API that can be used to develop web applications.

APPENDICES

APPENDIX A

ACRONYMS AND ABBREVIATIONS

ABET	Accrediting Board for Engineering and Technology
API	Application Program Interface
CRUD	Create, Read, Update, Delete
CSCI	Computer Software Configuration Item
CSS	Cascading Style Sheets
CVS	Concurrent Versions System
DHH	David Heinemeier Hansson
DRY	Don't Repeat Yourself
ERB	Embedded Ruby
HTML	HyperText Markup Language
IA	Information Architecture
JS	JavaScript
KSC	Kennedy Space Center
LCS	Launch Control System
MVC	Model, View, Controller
NASA	National Aeronautics and Space Administration
NE	NASA Engineering

PDF	Portable Document Format
REST	Representational State Transfer
SCC	Source Code Control
SCSS	Sassy Cascading Style Sheets
SSH	Secure Shell
SQL	Structured Query Language
SVN	Subversion
TOM	Task Order Manager
URL	Uniform Resource Locator

APPENDIX B

CONTRACTOR SURVEY SCREENSHOTS

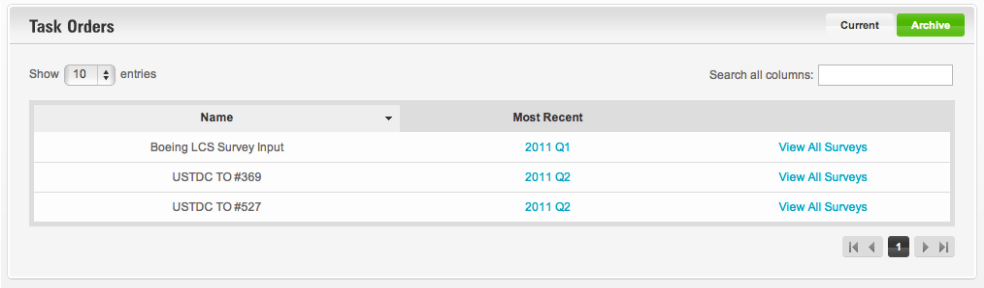

Figure B-1. Task Order List.

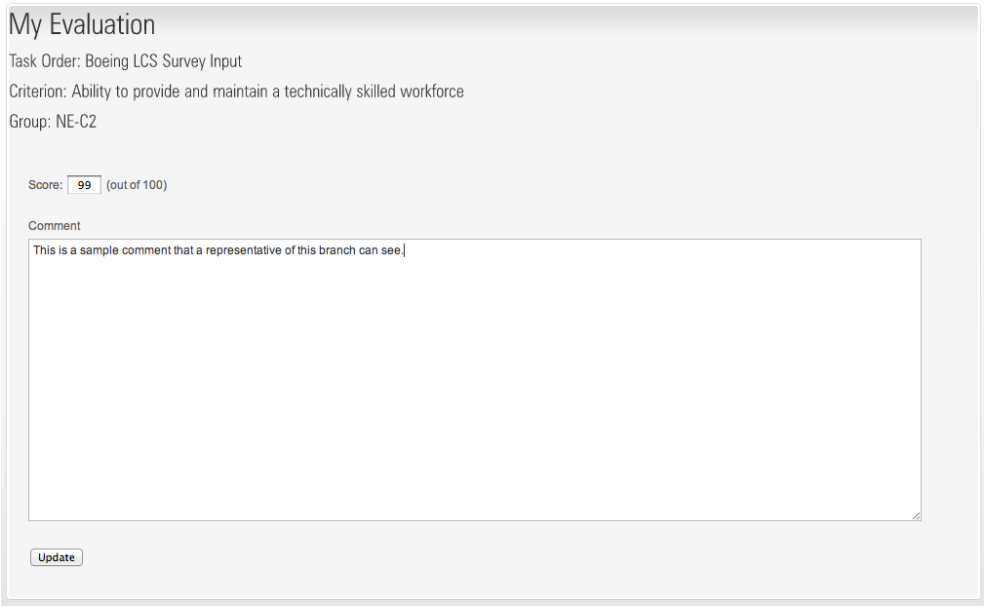

Figure B-2. New Evaluation

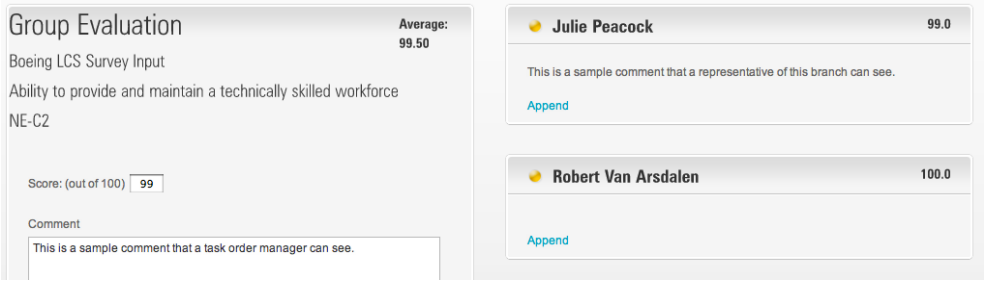

Figure B-3. New Group Evaluation

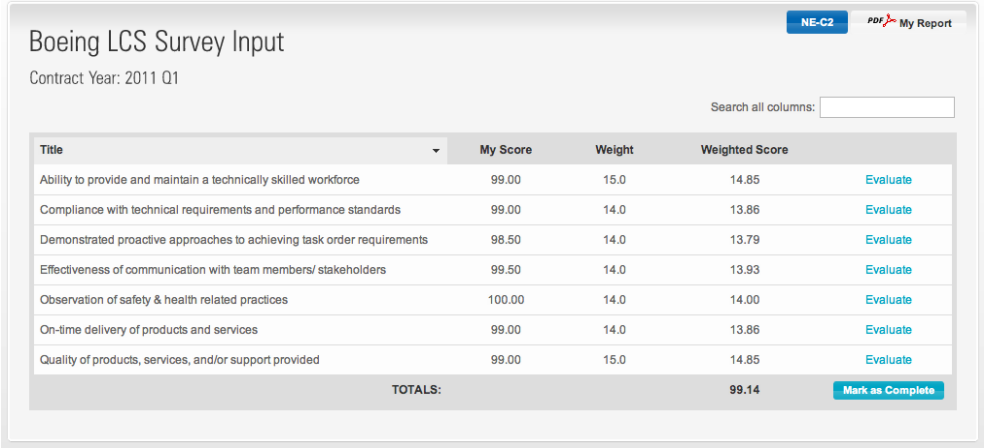

Figure B-4. Group Survey

APPENDIX C

RELATIONSHIP TO COMPUTER ENGINEERING PROGRAM OUTCOMES

This appendix is designed to display that this Thesis Project meets the required program outcomes for Kettering University's Electrical and Computer Engineering Department and the Accrediting Board for Engineering and Technology (ABET). Each of the program outcomes are listed with a brief explanation of their relevance to this project.

Program Outcome 1. Assembly language. Analyze, design, develop, debug, and document structured assembly language programs for at least two different embedded-computer platforms, including at least one with a 32- or 64-bit architecture. Use appropriate techniques and modern embedded-computer development tools.

The requirements for this Thesis did not require the Author to analyze, design, develop, or debug using assembly language.

Program Outcome 2. High-level language. Analyze, design, develop, debug, and document programs in at least one structured high-level programming language. Use appropriate techniques and modern software development tools.

The primary high-level language used throughout this entire Thesis Project was Ruby. This project also demonstrates the use of JavaScript.

Program Outcome 3. Real-time operating systems. Develop, debug, and document a simple real-time operating system and design, develop, debug, and document application programs for it to implement a complete real-time system that meets specifications. Use appropriate techniques and modern embedded-computer development tools.

The requirements for this Thesis did not require the Author to work with a Real-time operating system.

Program Outcome 4. Analyze, design, prototype, debug, and document combinational and sequential digital circuits. Use appropriate techniques and modern digital-systems development tools and implementation technologies.

The requirements for this Thesis did not require the Author to analyze, design, prototype, or work with digital circuits.

Program Outcome 5. Computer architecture. Design and verify the operation of a basic central processing unit for a general-purpose computer. Use appropriate techniques and modern digital-systems simulation tools.

The requirements for this Thesis project did not require the Author to design or verify the operation of a basic central processing unit.

Program Outcome 6. Circuits, electronics, and systems. Model, analyze (at DC and AC steady state), and design electrical and electronic circuits and systems. Use modern electronic design and test equipment.

The requirements for this Thesis project did not require the Author to work with circuitry or electronic systems.

Program Outcome 7. Elective areas. Use understanding of basic principles and appropriate tools to analyze, design, develop, debug, and document simple systems in at least two of the following areas of computer engineering: computer networks, programmable logic controllers, expert systems, database systems, VLSI systems.

This Thesis Project heavily involved the understanding of basic principles and appropriate tools to analyze, design, develop, debug, and document database systems. Since this application is a web application, and a great deal of information needs to be stored for each user, a database was the perfect choice. Ruby on Rails also utilizes the database of the developer's choice.

Program Outcome 8. Teamwork. Work productively in a multidisciplinary team, in particular to carry out projects involving computer engineering.

This Thesis Project was mostly designed and worked on in an individual setting, but there were times when the Author worked closely with his mentor as a small team. When working as a team, many design decisions needed to be made in order to develop a successful web application.

Program Outcome 9. Ethics and professionalism. Act in a professional and ethical

manner in the workplace.

As a Civil Servant and NASA Engineer, there are many professional and ethical standards that must be met. When working with a database system, it is essential to act appropriately keeping ethics in mind. This particular database contained information regarding specific contractor evaluations. The Author needed to ensure that only NASA personnel can see the evaluations and to be careful when using the information.

Program Outcome 10. Written and oral communication. Communicate effectively through written reports and oral presentations appropriate for other computer engineers or for non-technical audiences, as required.

This Thesis Project required a vast amount of oral communications. The Author met reguarly with one of the NASA Engineering Division Chiefs to make changes to the requirements and get requests for new features. In addition to the scheduled meetings, there were multiple demonstrations of the finished web application to upper management as well as a separate team in the KSC Headquarters Building interested in utilizing this product.

Program Outcome 11. Global and societal context. Understand the impact of engineering solutions in a global and societal context.

This Thesis Project did not require a global context, but the understanding of the impact can be found in a societal context. The project could eventually roll up to be agency wide.

Program Outcome 12. Lifelong learning. Independently acquire the information and understanding necessary to complete projects or undertake other responsibilities in unfamiliar areas from appropriate sources such as books, training courses, technical documentation, and application notes.

The Author needed to research and understand Ruby on Rails very thoroughly, so the Author took many training courses to gain a better comprehension. For example, to effectively understand testing the Author

proceeded to read through the Cucumber book. The Author also attended training and gained experience in Ruby on Rails outside of the workplace on a regular basis.

Program Outcome 13. Contemporary issues. Understand contemporary issues, especially as they relate to employment as a computer engineer.

This program outcome does not relate directly to this Thesis Project.