Agilent Technologies, Inc.
TCC Programming Search Interface

TJ Taylor

Daniel Johnson

June 23, 2011

Contents

Abstract 4
Introduction 5
System Design 6
  Functional and Non-Functional Requirements 6
  Architecture 7
  Design for the Spider 7
    Database Schema Definition 10
  Design of the Search Interface 12
Results and Implementation Details 13
  Why Ruby 13
  Why Ruby on Rails 13
Conclusion 14
  Areas for Expansion 14
  Problem Areas with the System 14
  Where to Find Help 15
  Glossary 15

Table of Figures

Figure 1: Architecture Diagram 7
Figure 2: Process Flow Diagram for Spider 9
Figure 3: The Database UML Diagram 10

Abstract

Over the years, Agilent Technologies, Inc. has produced a large collection of sample automation programs and program snippets that are used to control the equipment that their company manufactures. Presently, this collection has grown unruly and very hard to search, and as such Agilent needs a method to categorize and catalogue the stockpile of sample programs they have accumulated.

The issues were resolved by building a back end Spider that was written in Ruby and front end that was written in Ruby on Rails. The Spider application, similar in concept to the spiders that Google uses to index their data, runs on the Windows Share Drives that contain the sample programs, and inputs the information into a relational database. The back end and front end, combined with a MySQL database form a system that enables anyone to quickly search the repository of sample programs based on some search criteria.

Introduction

Agilent Technologies, Inc. produces radio frequency (RF) and microwave network analyzers, as well as various other types of measurement equipment. This equipment ranges from the small scale to devices that can cost upwards of $100,000. Over the years, Agilent has produced a large collection of sample measurement automation programs and program snippets, stored on a Microsoft Share Drive, that are used to control the equipment that their company manufactures. The problem is that these shared drives have little to no logical file structure and it is very difficult to find necessary programs. Often only a few people are relied on to find the necessary programs and in the worst case the program cannot be found and is rewritten, assuming that it already exited. This creates inefficiency in the company and possible loss in business due to frustrated costumers.

The client's first and primary requirement was a system that allowed for fast, easy, logical searches of the data stored in their Microsoft Share Drives. The implementation chosen consists of a Spider program, written in Ruby, to parse the data on the drives, and a web front-end, written in Ruby on Rails, to display and allow for searches of the information.

System Design

Functional and Non-Functional Requirements

- Spider
  - High level - The system must accept a list of root files, languages/extensions, and return a list of sample programs under those roots along with their related files and relevant information.
  - Inputs
    - A databased list of file directory roots where the spider should begin searching.
    - A databased list of programming languages and their extensions for the spider to look for
    - A (potentially empty) databased list of file, directories, and sample programs for the spider to update
  - Outputs
    - A databased list of sample programs, along with their related files (images, text, pdf, etc.) and relevant information
  - Lower level
    - The system should keep track of where a file should be located.
      - By back-referencing the database, the spider will be able to flag the files that have been moved/deleted.
    - The system should obtain relevant information about file
      - By cross referencing the pieces of the directory structure with the fields in the database, the system should be able to determine the related language, product family, etc. about the sample program.
    - The system should be able to ascertain which files are related to a given sample program.
      - By looking at similar file names and extensions, the system should be able to grab most of the related files for a given sample program.
    - The system's run time will be controlled by the Windows Task scheduler.
      - The spider will run at a client-defined time interval set in the Windows Task Scheduler GUI. This particular interface allows several options to the user and also provides notifications to the user when errors occur. Primarily, this interface allows the user to define how often the spider should run, and whether the program should be allowed to run several instances of itself.
- Search Engine
  - The system must allow for user-friendly searches
    - Upon acceptance of user input, among various fields, including but not limited to: Product Line, Product Number, Language, File name, and Keyword, the search engine should return a list of full file paths that show the locations of any files that match the criteria given by the user.
  - The system must allow login capability
    - Only system admins require logins.
    - Only system admins should be able to create users.
  - The system must allow for editable data
    - The user, with appropriate access permissions, should be able to edit the information presented in the search engine and that data should persist over various runs of the spider.
  - The system should notify users with appropriate permissions when a file has been moved/deleted.
    - When a file has been flagged by the spider in the database, the system should have some method of notifying admins that the file's location has been changed.

Architecture

Agilent provided a Windows workstation and an Ubuntu server for the system. The diagram below, Figure 1, shows the design that was used to fully implement the stated requirements. The Spider, residing on the Windows workstation that has access to the share drives, crawls through the shared drives and pushes the data onto the MySQL database located on the Ubuntu server. Then the Search Engine Interface, running on the Ubuntu server, uses the database to display sample programs to the users.

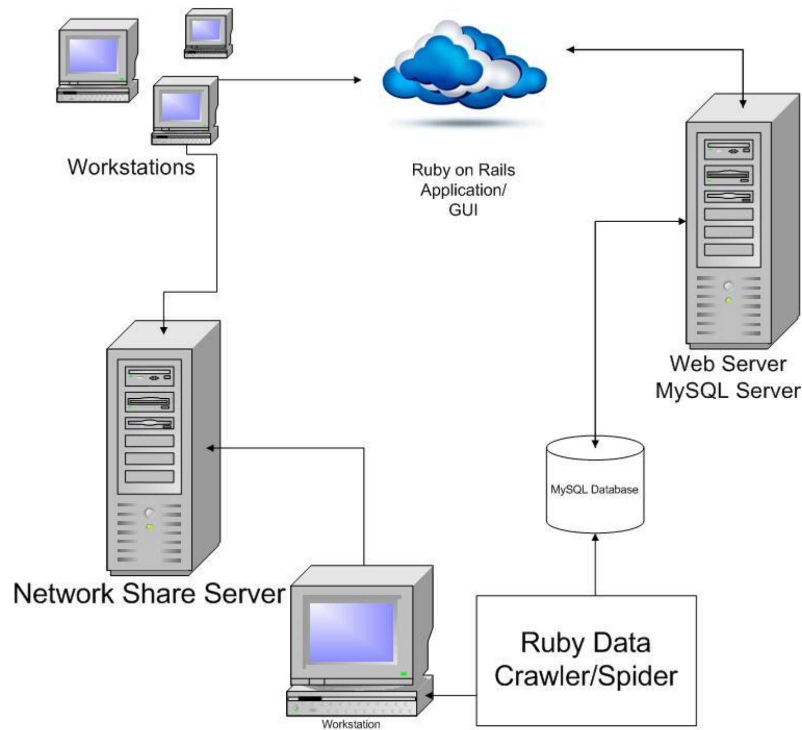

Figure 1: Architecture Diagram

Starting at the top left, moving counter-clockwise, the components are the windows workstations, the Network share driver server, the workstation that the Spider resides on, the Ruby Spider application, the MySQL database that resides on the Ubuntu server, as well as the Rub on rails application.

Design for the Spider

Spidering

Spidering is the process of methodically browsing through a set of data and returning some sort of relevant and useful information. Search engines typically include a 'spider' or 'web crawler' as part of their back end to accumulate and index all of the data that they use to generate relevant searches.

A history of spiders and web crawlers can be found on Wikipedia: http://en.wikipedia.org/wiki/Web_crawler. According to the Wiki page, it is known that spiders crawl through only a portion of the web, simply because of the sheer volume of data. Our spider however, must crawl through all of the data provided to it, and return some form of useful, user-interpretable information.

The Spider Program

The Spider program is a Ruby script that crawls through folders and files on a set of directories stored in the database. This is similar in concept to the spiders that Google uses to index web pages for their search engine.

The Spider runs daily to minimize lag between GUI and the physical realization on the server. For the most part, the files and programs are organized in the file structure such that a lot of relevant data for a particular sample program can be gleaned from its path. The Spider will take advantage of this layout by holding the current path in memory as it traverses the directory structure and breaking it apart into relevant data. That being said, the spider can only use what it knows. For instance, if the product that a sample program relates to is not in the sample program's file path, the spider cannot glean this information and will leave that field blank (or in some instances, obtain the wrong data).

The Spider will then update or feed this information into the database. If a new file is encountered, a new row is inserted into the table. If a file is no longer in its previous location, it will be flagged for inspection by the database admins.

To help with understanding the process the Spider has to follow, a Process Flow diagram was created to illustrate the information. This diagram is included in Figure 3, below.

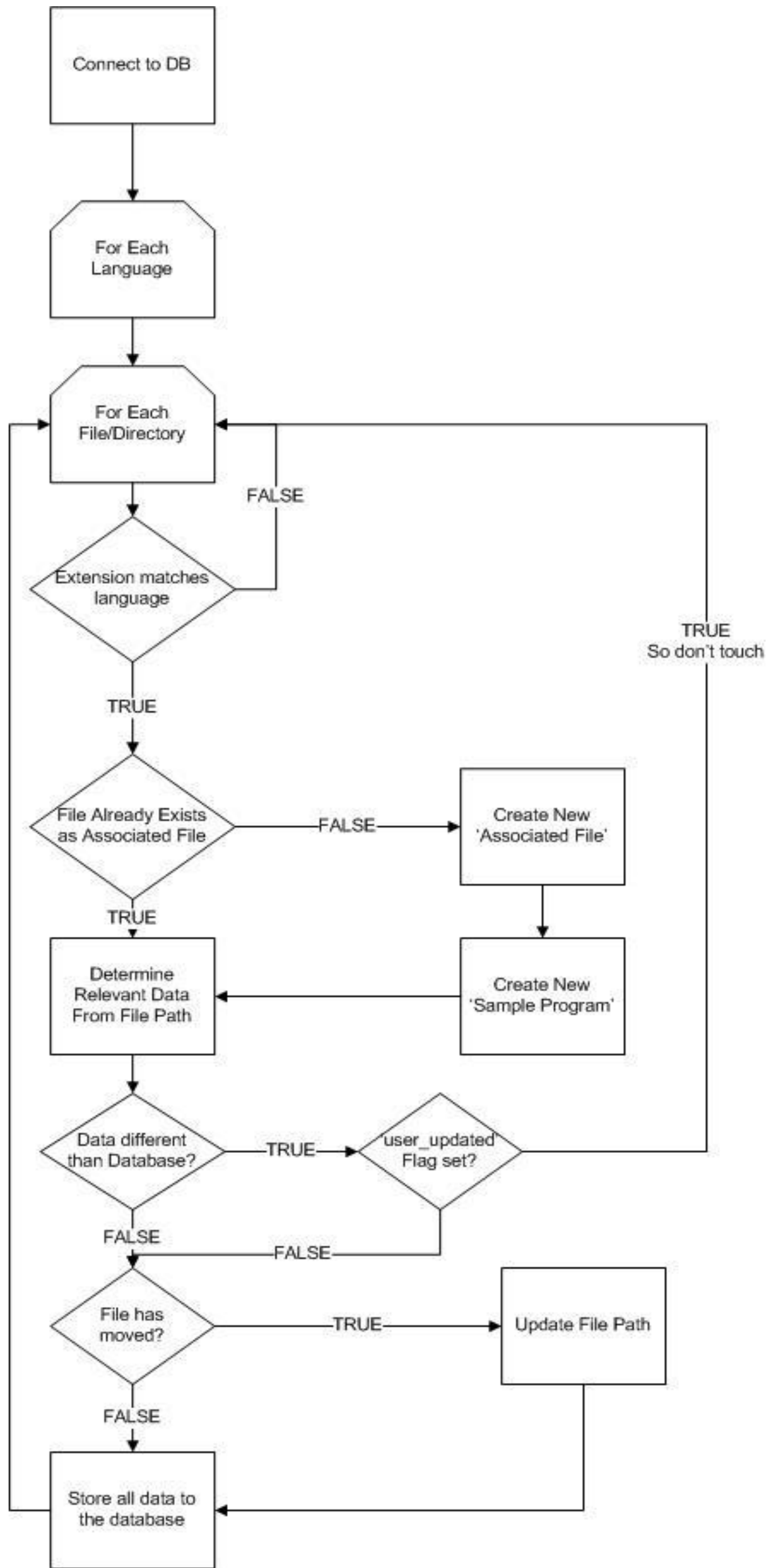

Figure 2: Process Flow Diagram for Spider

For any further information regarding the Spider, you can refer to the TCC Programming Search – Spider – Programmer's Guide or the TCC Programming Search – Spider – User's Manual.

Database Schema Definition

The database schema is below and is labeled as Figure 2. The schema focuses around the sample_programs table. Though the only key information that the sample_programs table contains is the name of the sample program and all other information is related to the table through foreign keys.

The table below the UML diagram describes the database schema in detail.

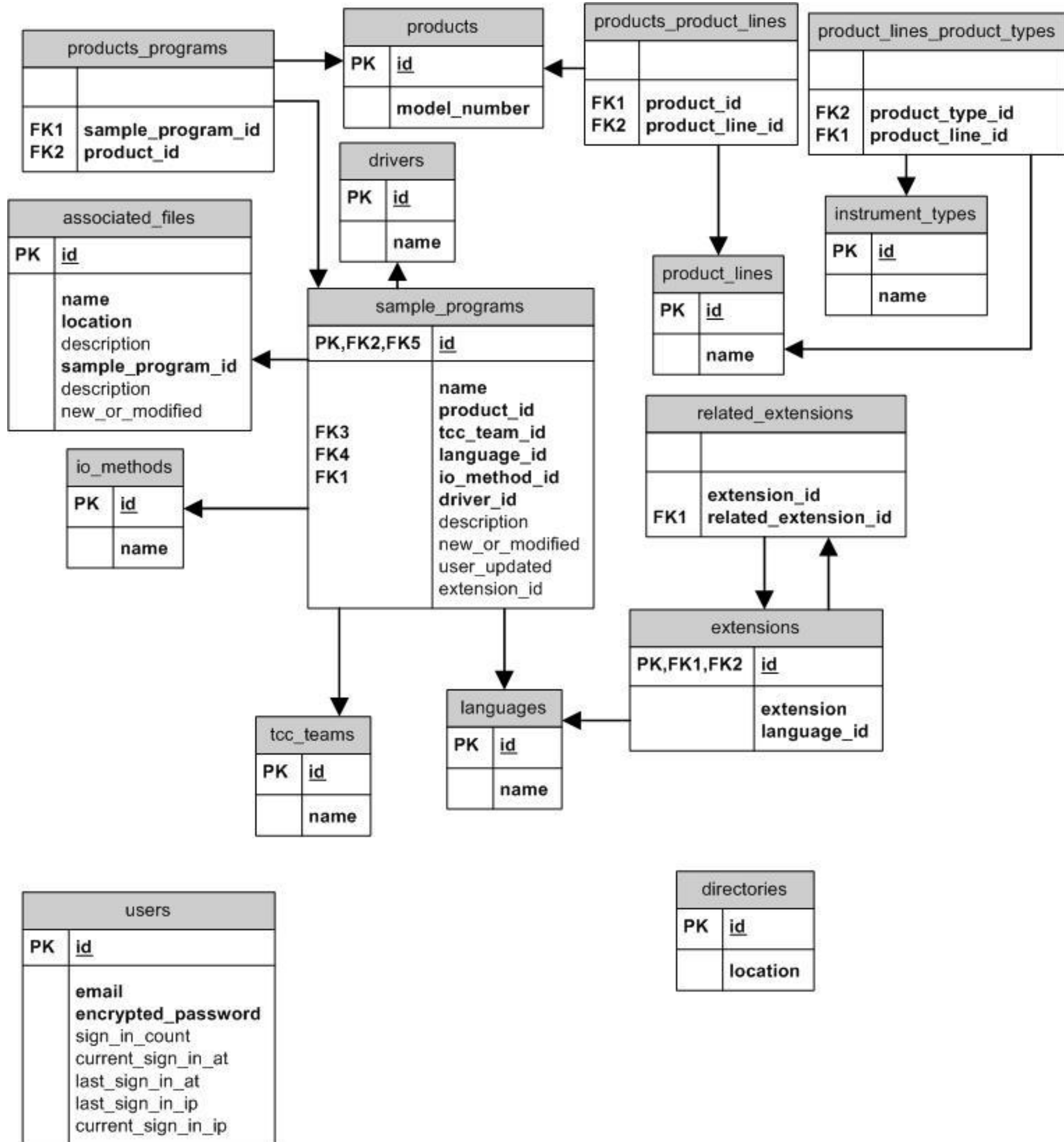

Figure 3: The Database UML Diagram

This table contains a majority of the tables in the schema diagram, located above in Figure 2.

sample_programs	The sample_programs table contains the sample program's name and several foreign keys. It also contains a description which will initially consist of data from the file path.
languages	Each row in the languages table only contains the name of a programming language, like Ruby.
extensions	The table 'extensions' contains an extension for a language and a language_id that points to a language. The language_id is only used if it is the main file extension for that language. For example, the extension 'rb' will contain the language_id for Ruby, while 'erb' will not because 'erb' is not the *primary* file extension.
extension_relations	To relate other extensions to a language the related_extensions table, the related_extensions table holds two ids that both refer back to a file extension. This linking of an extension to another extension allows the spider to know that an 'erb' file is related to a 'rb' file, for instance.
tcc_teams	The tcc_teams table contains the name of the team that created the sample program.
io_methods	The IO Methods table contains the names of io_methods that are used in sample programs.
drivers	The Drivers table contains the names of drivers used in sample programs.
associated_files	The associated_files table stores the location of all the files in the share drives, along with a foreign key relating that file to a sample program.
Products	The products table contains product numbers.

Design of the Search Interface

The Search Interface is a web application, written in Ruby on Rails, that allows for quick searches of the data created by the Spider. The interface also allows for editing the information found by the Spider, as well as some minor reporting on the data. The system was left open enough that this reporting functionality can be easily extended by the client.

Ruby on Rails uses many 'Ruby-isms' and programmatic principles and patterns such as DRY (Don't Repeat Yourself) and "Convention over Configuration." Convention over configuration takes a lot of the work off the programmer by saying that if you follow conventions set by the framework, your life is made much easier. One example of this is ActiveRecord, a collection of Ruby classes and modules that make up an ORM (Object Relational Mapping) for handling relational databases.

However, not everyone in the company should have write access to the database, so the application had to allow for some method of user authentication. This was handled through use of the Devise gem. A gem is a software package for Ruby, much like a module in PERL. The Devise gem creates all of the models, views, and controllers needed for handling user authentication and generates it for a specific domain with minor configuration and only a few simple command line calls. What the client decided was to only create users for administrators; everyone else has access only to the search functionality.

The searching functionality was handled by the Sunspot gem, which interfaces with the Solr Engine. The Solr Engine is a program that constantly indexes the data in a database for use by an application. This way, the database is not handling the heavy lifting of indexing the data, and only has to serve up the information that Solr tells it to. The search capability could have been handled by writing SQL statements that corresponded to what the user input on the search page, however, this would have require a lot of time to write these statements and it was decided that using a pre-existing library would be better.

The majority of the code base was created by Rails' built in *generators*. These are command line tools that create boilerplate HTML and ERb (a type of Ruby template software) that worked for many of the functions of the search interface. For instance, running 'rails generate scaffold Product name:string' at the command line interface(CLI) generated not only the HTML views, but also the controller for the HTTP requests as well as the database models and migrations to create the table in the MySQL database. This cut a lot of time out of our development.

There were some interesting roadblocks along the way, though. One such roadblock was the relational mapping from an extension to an extension. This was a mental obstacle at first, how do you relate a Language to all of its possible file extensions? The solution chosen involved setting a primary extension for a language (ex. '.c' for C, or '.cpp' for C++), and then having a join table that related that one extension to all of its related extensions. The solution is simple in its conception, but much more difficult in its implementation. Ruby on Rails apparently was not set up for this kind of database back-door work, and threw a fit every time we tried to set up this associativity. Eventually, we managed to get it working by creating a relationship from a file extension to another file extension and using a lot of the configuration that Rails is not fond of.

For any further information regarding the Search Interface, you can refer to the TCC Programming Search – Search Engine – Programmer's Guide or the TCC Programming Search – Search Engine – User's Manual.

Results and Implementation Details

The Spider and the Search Interface have both been successfully installed on the client's hardware and are running smoothly. The Spider is running once per day and the Search Interface is currently servicing search requests.

With our current implementation, the Spider takes approximately six hours to run through all of the data on the share drives. If the client chooses, they may increase the number of runs per day for the Spider, to further minimize the lag between what is on the share drives and what is in the database.

The Search Interface is running on the WEBrick server off the Ubuntu machine. WEBrick is a lightweight, Rails-compatible server, written in Ruby, that comes packaged with Rails. This made it a perfect choice for the web server because it will not interfere with anything else on the Ubuntu machine, and there was no additional installation required.

Why Ruby

Ruby was the language of choice for this project because of its simplicity and programmer friendliness. Because of Ruby's almost English-like syntax, the code itself can provide documentation supplemental documents. Also, one of the primary reasons Ruby was chosen was because of the ActiveRecord library. This library provides a dynamic, non-domain-specific Object-Relational Mapping (ORM) for Ruby that allowed our code to very simply connect to the MySQL database provided by the client and do all of the work with very minimal SQL coding.

The second reason for choosing Ruby was its very fast learning curve. For anyone familiar with scripting languages, Ruby provides all the flexibility they are used to, plus the structure and power of Object Oriented Programming.

Why Ruby on Rails

Ruby on Rails is a web application framework based on Ruby-isms like: "Convention over Configuration," and "Don't Repeat Yourself." The Rails framework follows the model, view, and controller architecture of development, and provides several very useful tools to make the programmers' lives simpler. Further, it allows for easy extension of the application, so that the client may add additional functionality to the Search Interface.

For any further information regarding the Ruby or Ruby on Rails, including a short tutorial, you can refer to the TCC Programming Search – Search Engine – Programmer's Guide or the TCC Programming Search – Spider – Programmer's Guide.

Conclusion

The goal of this project was to produce a working system that allowed the client to search their Share Drives. This goal has been achieved and the system has been deployed on the client's architecture.

Areas for Expansion

The client has mentioned a few additional features that would enhance the functionality and usefulness of the tool. These have been included here for completeness.

- Ability to generate an email for sending that any sample program, along with its associated files, to one of Agilent's clients.
- Further reporting functionality.
- More machine learning on the part of the Spider. Allowing the Spider to intelligently know what a piece of data is beyond what it has in the database.
- More dynamic searching on the Search Interface. The client has expressed a desire for an interface similar to Google's new AJAX search where results are displayed in real-time as the user selects search criteria.

Problem Areas with the System

- The Search Interface is written using HTML5.0 and javascript. Internet Explorer (IE) has known issues with both. These issues are compounded when earlier versions of IE or used (primarily IE7). When the upgrades are made to newer versions of Internet Explorer, the issues should be fixed.
- Both the Search Interface and the Spider are on machines that can be turned off at random times. This is generally uncontrollable by the client. This is obviously more of a problem for the Search Interface, as availability will completely disintegrate. Also, if the Spider's machine is shut off, it can create a large discrepancy between the data on the Share Drives and the database.
- WEBrick, the server being used for the Search Interface, though great for small scale applications, can be a resource hog when a lot of concurrent users are accessing the application. This may pose a problem for the client later on, but for now the client is not worried.

Where to Find Help

Help for the application can be found in any of the guides provided with this document, that is, any of the following:

- TCC Programming Search – Spider – Programmer's Guide
- TCC Programming Search – Spider – User's Manual
- TCC Programming Search – Search Engine – Programmer's Guide
- TCC Programming Search – Search Engine – User's Manual

In addition to these documents, several online resources are available for help. A few have been listed below.

- http://guides.rubyonrails.org
    - General information regarding Rails and it's components
- http://www.ruby-lang.org
    - General Information regarding Ruby
- http://www.mysql.org
    - General Information regarding MySQL
- http://www.rubyonrails.org
- www.stackoverflow.com
    - Searchable Forum for programmers

Glossary

- ***Devise*** – A Ruby gem that handles the difficulties of user authentication in Ruby on Rails applications.
- ***Gem*** – A software package for Ruby. Similar to libraries in C and Java, and modules in PERL.
- ***Find*** – A Ruby module that handles directory crawling.
- ***MySQL*** – A free, open-source, relational database.
- ***Ruby*** – A powerful, fast, object-oriented scripting language, with its base written in C.
- ***Ruby on Rails*** – A web development framework that makes web programming easier and more fun for the programmer.
- ***Solr*** – An engine that handles the indexing of information, similar to a database, but that handles queries more like those received from a search engine.
- ***Spider*** – A program that, for our sake, crawls through a directory structure (though it could be any data structure) and retrieves relevant data regarding that structure and its contents.
- ***Sunspot*** – A Ruby gem that interfaces with Solr.