Shayne N. Muelling. A Web-based Electronic Portfolio System for Information and Library Science Students: Project Report. A Master's paper for the M.S. in I.S. degree. May, 2009. 40 pages. Advisor: Robert Capra

This paper describes the design and development of a web-based electronic portfolio system for students enrolled in UNC’s School of Information and Library Science. The goal of the system is to provide students with an electronic space to collect and present projects completed during an academic program. This paper outlines the system requirements and technical specifications, and includes a description of the development process.

Headings:

Information Systems - Design

Computer Software - Development

Education - Information services

Computers - Information services

Portfolio - Electronic

A WEB-BASED ELECTRONIC PORTFOLIO SYSTEM FOR INFORMATION AND LIBRARY SCIENCE STUDENTS: PROJECT REPORT

by
Shayne N. Muelling

A Master's paper submitted to the faculty
of the School of Information and Library Science
of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements
for the degree of Master of Science in
Information Science.

Chapel Hill, North Carolina

May 2009

Approved by

_____

Robert Capra

Table of Contents

Introduction

Electronic Portfolios (e-portfolios), defined as an electronic collection of student work and reflection during an academic program, have been widely studied across a variety of disciplines in the past decade. In general, e-portfolios are seen as an excellent way to demonstrate student knowledge, skills, and abilities, and a possibly more effective alternative to traditional paper-based final examinations.

Final evaluations are an integral part of academic programs, for both students and faculty. The final evaluation process at its best provides a positive reflection for both students and faculty members. Final evaluations allow students to demonstrate to faculty members that they have gained the required knowledge to become proficient in their chosen field, which is the ultimate goal of a given academic program. Final evaluations can also be useful to students after graduating from the academic program, as a way to show potential employers the breadth of their knowledge.

Electronic portfolios have proven to be an excellent method of final evaluation for students as they provide a complete view of the work accomplished by a student during an academic program. Appropriately structured e-portfolio systems provide components that give students space to represent, revise, and reflect on academic work for the final collection. The combination of these three facets shows the full extent of discipline-specific skills, critical thinking skills, and presentation skills acquired by students during an educational experience.

The goal of this project is to create a prototype web-based e-portfolio system that will allow students to present many different types of projects completed in an information/library science degree program. The system will initially provide functionality aimed at giving users an application that will provide electronic space to collect projects and make them publically accessible to employers, with a development focus on extensibility in order to allow for a more fully featured system in the future.

Literature Review

There's a broad range of literature on e-portfolios tools for students. Many articles discuss the purposes of e-portfolios, like Ritzhaupt and Singh's study regarding student perspectives on e-portfolios (2006). The authors discuss the ways that e-portfolios are beneficial, including the way they increase learning effectiveness and model professionalism for students. The study also cites current literature on e-portfolios to define their purpose as a combination of “three R’s”: representation, reflection and revision. E-portfolios are a *representation* of student work, meaning they are a way to present collected student work. The process of collocating and describing each project encourages *reflection* by the student. The presentation of e-portfolios to faculty results in feedback, which often stimulates *revision* of the items in the portfolio. The study goes on to ascertain that an e-portfolio as a good representation of work accomplished by the student due to these elements. Ritzhaupt and Signh go on to apply these components as measures of student perspectives on the systems.

Fitch et al describes some of the advantages of e-portfolios as shown by previous research: increases learning effectiveness, models professionalism, enhances information technology skills, allows for academic credit for learning beyond the classroom, and

allows for reflections on artifacts as well as how they match goals and standards (2008). The literature available on e-portfolios suggests that they are a useful system for student reflection and evaluation. Additionally, the idea that portfolios model professionalism suggests that they are likely useful to students during both their academic career and their professional life.

There is also information directly related to the use of e-portfolios in technology-based academic fields. Bhattacharya, Heinrich and Rayudu studied the usefulness of e-portfolios in the academic fields of computer science and engineering (2006). For this study, the authors developed what they refer to as an "Open Source Portfolio" system, which is built upon a framework developed to categorize the type of skills a student gained and/or exhibited for each project. The categories (annotation sets, as the authors describe them) attempt to express life-long learning skills rather than specific technologies (i.e. "problem-solving", "team work"). The authors suggest that students begin with the graduate profile for their course of study, which lists all the skills a student should gain during their studies, and their own personal goals. From this point, they recommend students identify the artifacts that demonstrate their knowledge in a specific skill area, and then reflect on these artifacts in order to build the e-portfolio. These guidelines combined with the skill rubric were then used to create sample e-portfolios. Interviews of industry professionals viewing these sample portfolios confirmed the value of the e-portfolio for employers – employers relished the opportunity to see a student's capabilities.

Much literature is also available on the final evaluation process, the majority of which focuses on traditional methods of evaluation. Of particular interest to this study is

Hancock's recent research on traditional pen and paper evaluation methods of graduate student versus performance-based assessment (2007). Hancock's study found that students exposed to performance-based assessment (i.e. portfolios) had higher scores on final examinations. Results indicate that non-traditional assessment models – e-portfolios for example – may be more effective than traditional models.

Concept and Objectives

The main concept of the system is to supply student users with electronic space to describe and reflect on projects completed during a SILS graduate degree program. The system will then dynamically present the collection of projects in an organized manner via a publically accessible URL, so that potential employers, professors, and other interested parties may view the student's academic work. The system is an e-portfolio builder web application. In order to achieve these goals, the system must provide a framework to build a portfolio of projects. Each project can be described by a student in both text and images through the administrative component of the system. From this student-provided metadata, the system can then dynamically render a publically accessible web page presenting all projects entered into the system by a given student.

Given the concept outlined above, system needs to fulfill two essential functions:

1. provide student users with a way to enter information into the system for multiple academic projects; and
2. render a presentation view of all projects for a certain student.

In order to implement the first function, the system must allow student users to create an account containing personal information, including full name and a unique username/password combination. In order to implement the second function, the presentation view must identify all projects entered by a unique user and dynamically create a publically accessible presentation view. Privacy is an important concern for personal information, and ideally the system will give a student user full control over public access to the presentation view of their portfolio.

The following sections define system-specific terminology and outline use cases for two distinct types of users of the SILS e-portfolio web application: the students creating portfolios, and the employers examining student portfolios as supplemental material to a resume. Ultimately, I envision two additional types of users: professors examining the final portfolio and grading either individual components or the complete portfolio as a final evaluation, and system administrators maintaining the web application. However these users require additional functionality that fell outside the scope of my project.

System Vocabulary

The components of the system will be addressed in the following manner throughout this paper:

Dashboard – the student management component of the system. The dashboard is the space providing students with the options to: login, logout, add/edit personal information, add/edit/delete projects, and view the complete portfolio.

Presentation view – the publically accessible view of a student's collection of projects. The presentation view is the completed portfolio, containing all projects entered into the system by student, organized by skill.

Skill – the specific technology or technologies employed to complete a project; programming languages like HTML, Java, C++, and classification systems like Dewey Decimal and Library of Congress are all considered skills.

Project – the text and image description of an academic assignment worked on and completed by a student during an academic degree program at SILS.

Student Use Case

Jennie is a Master's student at UNC SILS who wants to add a recently completed database design project to her e-portfolio. She logs into the system and is presented with an administrative dashboard, which provides options to take different actions. Jennie clicks the "create new project" link on the dashboard, which loads a new page that displays fields to name the project and describe the project in free-text. Jennie titles her project "Pet Store Database Design" and selects the course(s) for which the project was completed from a given list of courses. She also selects skills that were gained or exhibited by the project from a pre-existing collection. She notices that "Entity Relationship Modeling" is not a skill in the list, so she clicks the "add new skill" link and completes the form appropriately. She submits the form, the new skill appears in the list, and she selects it in addition to the other skills she's already selected.

After entering the text information to describe the project, Jennie submits the information and reviews it. She also wants to add a digital image of the Entity-Relationship diagram and a Relational Schema she created for the project. Jennie clicks

the “add file” link at the top of the “Review Project” page and sees a form for uploading a file. She uses the form to browse to the local file on her machine, captions it “ER Diagram” and clicks the submit button. The file immediately appears on the “Review Project” page, under the “Files” subheading. As she’s reviewing the project, Jennie notices a misspelling in the project description, and clicks the “edit” link to correct the error. Resubmitting the project allows her to reexamine the information.

Jennie decides she happy with the project description and associated files, but she’d like to see how this project appears in the presentation view. She clicks “back” to return to the administrative dashboard, and then clicks the “Portfolio View” link in the left-hand menu to load the presentation view. Jennie sees how the newly created project appears to viewers of the portfolio. From the presentation view, she clicks another link to return to the dashboard. Jennie realizes she’s late for class and logs out of the system through the dashboard, returning at a later time to add more projects, edit previously created projects, or edit personal information.

Public Use Case

Tom is the hiring manager for XYZ Databases. He recently received a job application from Jennie, the SILS student from the previous section. Jennie’s resume includes a URL to an e-portfolio. Tom opens the location with a web browser and loads Jennie’s e-portfolio. He is presented with a web page displaying Jennie’s contact information and a menu containing a list of skills. Tom is hiring for a position requiring a strong understanding of database design and maintenance. He notices “Entity Relationship Modeling” in the list of Jennie’s skills and he clicks it, expanding a list of projects Jennie has entered with this skill. He clicks on the first project in the list, “Pet

Store Database Design" and the metadata for that project loads in a pane in the page. Tom reads the project description and examines the ER diagram that Jennie has uploaded with the project so the employer can see exactly what the project consisted of. Tom examines several other database projects in Jennie's portfolio. He is impressed by her knowledge and experience with database design and modeling. Tom clicks on the "profile" link to access Jennie's email address in order to contact her to schedule an interview for the position.

Initial System Description

For this project I continued working on a web-based e-portfolio system developed during the Fall 2008 SILS course INLS 672: Web Development II. This project was a collaborative effort between Erin White (MSIS 2009), Kyle Richardson (MSIS 2009), and myself. I obtained permission from my partners to expand the implementation of our shared project. The system contained the following components/functionality at the start of my individual development:

User registration/login functionality:

The root page of the application presents users with the option to "Register" or "Login". Clicking the "Register" link loads a form with required username and password fields and optional additional personal information fields (full name, email address, mailing address).

Clicking the "Login" option loads a login form where a user enters a username & password. Submitting the data validates it against the database - successful validation allows a user to access the dashboard of the system.

Add/edit/delete project metadata:

A user can add a project with the following fields: project title, skills and courses associated with a project, and project description. A user can also edit these fields on a project, and delete a project, removing it entirely from the system.

However, this iteration of the system does **not** support uploading images.

Course metadata sanitized and extracted into the application's database:

The only readily available listing of all SILS courses – including course numbers, titles, and full descriptions – is on the SILS website encoded in HTML (http://sils.unc.edu/programs/courses/descriptions.html). The number and title information for each course was extracted and cleaned up (removing all tags and extraneous data) from this .html file using a Perl script, then loaded into the database using a MySQL script.

All students' projects are visible without a way to securely filter only a single student's portfolio:

The presentation view of this iteration of the system has an index page that lists all students in the system – clicking one of the links loads that student's portfolio. Each portfolio has a link back to the index on the bottom. Users visiting the presentation view have access to all student portfolios.

Horizontal bar of tabs across the top of the presentation view, accessing student projects organized by the "type" of project:

Project type is a very vaguely defined field in this iteration. Ideally, we imagined that "project type" would be categories like Databases, Web

Development, Cataloging. But a controlled vocabulary for this field was not developed and consequently it is a free-text field without a concrete definition in the application, making it a poor choice of field on which to organize portfolios.

Presentation view displayed with aesthetically pleasing CSS:

The CSS for the presentation view renders the content with a consistent and well-designed “theme” (look and feel).

Objectives and Goals for Expanded System

I identified other components and functionality that needed to be added to the system to fulfill the requirements of users in a meaningful way. The list proved to be prohibitively long, so I prioritized several key elements. Elements from my list that I did not develop are listed in the “Future Directions” section of this document. The following is the collection of elements that I successfully integrated into the prototype:

Student users are able to upload images:

The system allows student users creating portfolios to upload multiple images to each project. Images that students may want to upload include: screenshots of web sites, system models, database models, etc. At this time, the system does not support pdfs, only standard image files including GIFs, JPGs, and PNGs.

Integrated AJAX functionality to two sections of the administrative back-end:

AJAX (Asynchronous JavaScript and XML) functionality allows elements on a webpage to be updated without having to reload the entire page. This provides a faster user experience, as the user doesn’t have to wait for the entire page to reload. I added AJAX to the “add skill” form in the new project page, so that the form appears on the page only when a user requests it. Ideally, adding a

new skill only updates the skill selection box on the new project page, although this functionality hasn't been fully implemented at the current time.

I also added AJAX to the "add file" form in the view project page, so that the form appears on the page when a user requests it. Adding a file to a project updates the view so that the image file is visible at the bottom of the rest of the project metadata, without reloading the entire page.

Redefined relationships between the many-to-many elements "project" and "skill" to reflect current best practice in the chosen web framework:

The Rails web framework (described in "Implementation Decisions") has two different many-to-many association types. The newer association type has less documentation, but is currently recognized in the Rails community as the more fully featured type of association. The older association has been unofficially deprecated by the community of Rails developers (see "Redefining Association Types" for more information).

Redesigned the presentation view to make it more logically organized and user-friendly:

I replaced the tab interface with a vertical expanding menu that allows users to see all project titles at once. I also changed the organizing principle from "project type" to "skill".

System Requirements

A student user must be able to register her/his identity by creating a unique username and password combination. After registering, the user must be able to log onto the system with their chosen username / password. The user needs to have access to the following basic actions once they are logged onto the system:

- edit personal information
    - password, name, phone number, email address
- create new project
    - add self-assigned metadata to a project including title, description, dates worked on, skills required to complete project, etc.
    - assign pre-existing metadata to a project (course information)
    - upload screenshots of the project
- edit existing projects
    - add images
    - change text metadata
    - delete images
- access the presentation view

The system must generate a URL of each user's complete portfolio that can be given to potential employers as a supplement to a traditional resume. The aforementioned requirements will provide students with the high-level functionality of being able to create a comprehensive electronic collection of academic work quickly and without laborious coding to achieve the end goal of rendering the portfolio in a web browser.

Students' portfolios can contain the following specific items, in addition to the basic text metadata:

- screenshots including:
    - static images of websites (jpg/gif/png)
    - diagrams/models including DB schemas, ER diagrams, use case models, etc (jpg/gif/png)
    - excerpts of papers (pdf)
- links to live websites (URLs)
- code snippets (txt)

The system will be extensible to allow for the possibility of additional file types and objects in the future.

The presentation view must be dynamically rendered from student user provided metadata. The presentation view must be organized well, so that an end user (i.e. potential employers) can access the information easily and without supplemental instruction. Additionally, the presentation view should be aesthetically pleasing and well organized so that the student's work is showcased in an effective manner.

Finally, the system must be secure. Only users who are logged into the system will be able to access the administrative portion of the portfolio. Users will only be able to access the administrative functions for their own projects. The passwords for users must be stored in a secure fashion, using a salted hash, and the system must securely send/receive password information to/from the database. The presentation view for a specific student will not provide access to other students' portfolios. These elements all help to ensure the security of personal data submitted to the system by student users.

Technical Details

Implementation Decisions

I chose to use the open-source Ruby on Rails framework to implement the e-portfolio application. This framework seemed ideal for several reasons. First, the Rails framework encourages a Model-View-Controller (MVC) development architecture. MVC architecture encourages best-practice development by isolating the business logic of an application from the user interface. Rails implements MVC by making it clear where different sections of code belong. The MVC architecture of Rails applications helps to form a code base that's easy to maintain and easy to pass along to other developers. The conceptual e-portfolio system I proposed is extensive. It was decided that due to the limitations of the scope of this Master's project that parts of the application would not be implemented (see "Future Directions"). The MVC architecture is extremely advantageous given these constraints, as it results in a cleanly coded application that is easy to pass along to subsequent developers.

Additionally, the Rails framework primarily excels at scaffolding database-driven web applications. I had already decided to implement the e-portfolio application with a database back-end. The information the e-portfolio application must be able to store and access (i.e. user, project, skill, and course information) is ideally suited to database storage. Rails contains powerful built-in methods that allow developers to quickly and easily create and maintain an application database from within the application code, eliminating the need to access the database directly. The lack of direct database access

makes it less likely for the application and database to get out of sync, and allows developers who aren't proficient in query languages to manage an application database.

Another reason I elected to use the Rails framework is that it's a technology that I was unfamiliar with prior to beginning this project. The Rails framework is coded in the Ruby programming language, and applications built on Rails are also coded in Ruby. Not only was Rails a new framework for me, Ruby was also a new programming language for me. Learning these skills during the construction of the e-portfolio system provided me with an opportunity to continue learning while practically applying my information science education to this point.

Technical Specifications

The e-portfolio application has the following technical specifications:

- Rails version 2.1.1
- Ruby version 1.8.5
- MySQL database, version 5.0.45
- Rails plugins: multiple select 20080608, state select, upload column
- Plugin dependencies: ImageMagick and RMagick (with associated dependencies)

I developed the e-portfolio application on a server running Red Hat Enterprise Linux 5. I configured the application to run on Apache Web Server software. The full source code is available at http://www.shaynemuelling.com/eportfolio/e-portfolio-20090429.tar.gz

Description of Relational Data

A registered user may have many projects. Each student user can have multiple projects associated with their unique username. This helps ensure that students use the system as it is intended – to collect many projects into a complete portfolio of academic work.

A project belongs to a single user. Although some projects may be developed in groups of multiple students, it's important for each student to describe the project in their own words in order to gain the full reflection and revision experiences that make creating an e-portfolio a useful learning tool.

A project may have many skills. A single project may exhibit mastery of many different skills; i.e. a Pathfinder created by a Library Science student may require cataloging skills and web development skills.

A skill may have many projects associated with it. Each skill in the system should be available to every user of the system. This lessens redundancy in the database.

A project may be associated with many courses. Although many projects will be for a single course, some projects may extend across multiple courses. This is particularly important for SILS graduate students, due to the nature of the courses and the population. For instance, the SILS "Systems Analysis" course consists of a semester-long group project that does not need to be implemented. However, many students build off the project for this course in subsequent, related courses. Additionally, some graduate students at SILS are working professionals who elect to use examples from their jobs for course assignments, and these may extend beyond a single course.

A course may have many projects associated with it. Many students take a single course and each student may have a project for that course. Courses are not identified by semester, although this data would need to be added to the system if it were used as a grading system for professors.

A project may have many images associated with it. Each project may have multiple image files connected to it, so that a project can have views of different aspects of a project. For example, a student may have screenshots of different web pages for a single web site project.

An image belongs to a single project. Each image stored in the web application can only be associated with one project. The images uploaded for a given project is unique to that project.

System Construction and Challenges

In this section, I will describe the development of the system components listed in the section titled “Objects and Final System Description”.

Upload Functionality

I decided to start with the component I had been struggling with at the close of the previous semester – file upload functionality. I began by thinking about whether I wanted to store the files in the database or on the file system. Both are equally viable and effective, and each has pros and cons. Storing files in a database requires using a binary data type for the field in the database that will hold the files. Most relational databases have a binary type, but storing binary data in a database rapidly expands the size of

database and space is finite. Also, accessing files from a file system has a small increase in retrieval speed. An advantage of storing binary data in the database is that it keeps all the application's data centralized. Ultimately, I decided to store user images on the file system. Storing files/documents/images on a file system is common practice in this type of system as it makes more logical sense to store files in a system developed expressly for that purpose.

Although Rails has many built-in file upload methods that make file uploads fairly easy, I discovered a file upload plug-in called "UploadColumn" that makes the process even easier. The plug-in includes methods to sanitize the uploaded data, uses RMagick and ImageMagick to implement easy resizing of images (important for creating thumbnails), and also has methods to specify the destination folder of images in the application (2007).

I began by installing the UploadColumn plug-in into the vendor/plugins directory of the portfolio application. Since I would not be storing the files in the database, I didn't need to create a table for the files in the database, but I would need a table to store the location of each image on the file system. I created a migration for the uploads database table, with the fields image:string, project_id:integer and caption:string. Given the one-to-many association between images and files, I needed to store the primary key from project (project_id) along with each file to enforce the association in the application.

Next I created the upload model, which is where I harnessed the power of the plug-in by using the built-in UploadColumn method "upload_column" to specify the data type of the field in the uploads model:

```
class Upload < ActiveRecord::Base
    validates_presence_of :image
    upload_column :image
```

```
    belongs_to :project
end
```

I had to make sure to name the “upload_column” in the upload model exactly the same as the field in the uploads table that stores the file system path (image). I also specified the relationship between upload and project through the “belongs_to” statement. The projects model has a corresponding “has_many :uploads” statement. Now the application was prepared to accept uploaded images for each project.

The real challenge came when I tried to add the upload option into the new project form. These types of complex forms are somewhat difficult to implement in Rails, because multiple controllers are being accessed and the associations between tables must be maintained. The latter proves difficult because the project_id is required to create a new upload object, and this id is not available until after creating the project. I looked around for some resources and discovered Ryan Bates’ “Complex Forms” Screencasts, which provided me with an excellent template for implementing a form with multiple objects (2007). However, I was unable to successfully implement the complex form in my application even after many attempts and much time spent troubleshooting.

I changed my tactic slightly, and settled on a slightly different implementation – instead of giving users the option to upload files **while** they create a new project, I give them the option to upload files **after** creating a new project. Although this wasn’t my first impulse, it actually makes quite a bit of sense, given that the two types of data (text metadata and image representations) are quite different. The project’s text metadata is assigned and submitted then the user is presented with a screen that allows them to examine the text metadata and a link to “Add Files”. The link expands using AJAX (explained in “AJAX” section below) to show a form for file uploads.

The code to add the file to the project exists in the uploads controller, and uses some of the built-in association Rails helper methods. I'm able to call upload through project using the @project.uploads syntax, since each upload must belong to a single project. Then I pass the upload information into the create! method and redirect the page back to the show project view with the newly uploaded file shown at the bottom of the page:

```
def create
     @project = Project.find(params[:project_id])
     @upload = @project.uploads.create!(params[:upload])
    respond_to do |format|
          if @upload.save
            format.html { redirect_to(@project) }
    . . .
end
```

The code for the file form in the project show view is fairly straightforward since the form is no longer complex:

```
<div id="fileForm" style="display: none">
<% form_for [@project, Upload.new], :html => {:multipart => true}
do |f| %>
      <p>
            <%= f.label :caption, "Description"%>
            <%= f.text_field :caption %><br />
            <%= f.label :image, "File"%>
            <%= f.upload_column_field :image %>
      </p>
      </p><%= f.submit "Add file" %></p>
<% end %>
</div>
```

The "upload_column_field" method is an UploadColumn plug-in built-in method, which takes care of sanitizing the uploaded data, storing the file on the file system, and storing the path to the file in the appropriate field in the database. It's necessary to set the form to allow multipart data so the file is uploaded instead of a string expression of the binary file.

I would have liked to integrate some of the resizing effects included in the UploadColumn plug-in, so that a thumbnail gallery could be presented to the user when

reviewing the project data. This functionality would also have made it easier to present images in the appropriate dimensions in the presentation view. However, my struggles with the complex forms set me back quite a bit and I had other components to attend to, so I did not implement any thumbnail or resizing functionality in the development version of the application.

AJAX Functionality

When I began developing the application independently, the "create new project" view listed the current skills with checkboxes, so that a user could select multiple skills to describe the project. The view also had a link below the checkbox, "Add Skill". This link loaded the "create new skill" view, and once the skill was created and submitted, the user was returned to the create project page. This flow seemed clunky to me – the user was taken away from the main goal of the process (creating a new project) in order to add an option to the list of metadata (a skill). It would be preferable to be able to add skills to the list without being taken away from the create project view. I wanted to use AJAX (Asynchronous JavaScript and XML) to be able to add a skill without reloading the entire page and without being taken away from the "create new project" view. I also didn't want the "add skill" form cluttering up the page for users who didn't need to use this option – it's confusing to see elements on the page that don't relate to every user's needs. I wanted similar functionality for adding files to projects, so that the file form wasn't always visible.

Ryan Bates' screencast explaining how to create a weblog from the Ruby on Rails site proved to be an invaluable resource (2008). In the screencast, Bates explains how to add comments to a blog post using AJAX – very similar to what I needed to do with

adding skills to a project. The first step was to enable usage of the built-in Rails JavaScript libraries in the views, by adding the following to the <head> section of the project layout:

```
<%= javascript_include_tag :all %>
```

Although it can make the pages a little sluggish to load all of the JS libraries available, I tested loading individual libraries and didn't notice much of a speed difference due to the (currently) small size of my application. Future iterations of the application may need to take this into account, but for now all the libraries are loaded so that I have maximum access for development.

It was fairly easy to implement the show/hide form functionality once the libraries were available. I simply put a div named "add_skill" in the "create new project" view:

```
<div id="add_skill"></div>
```

Then I created a partial view containing the form for adding a new skill:

```
<% form_for :skill, @skill, :url => { :action => 'update_project'
} do |f| %>
            <p>Skill: <%= f.text_field :skillName %></p>
            <p>Type: <%= f.text_field :skillType %></p>
            <%= submit_tag "Create" %>
<% end %>
```

Next, I created the link to the partial view using some JS effects from the scriptaculous/prototype libraries:

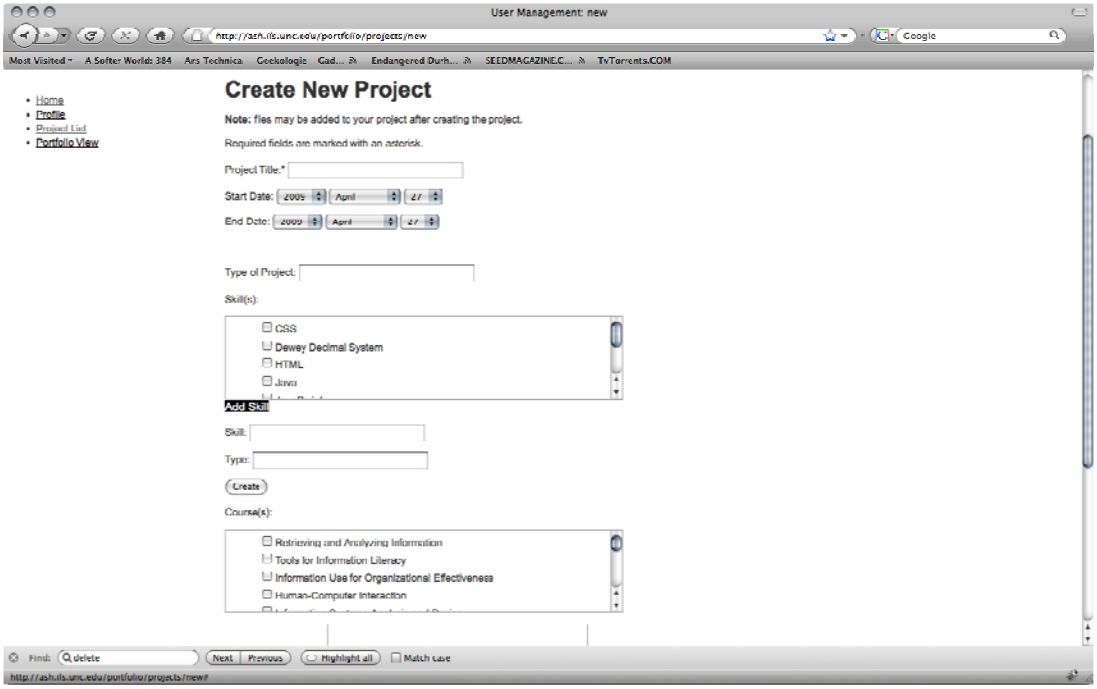
```
<%= link_to_remote 'Add Skill', :update => 'add_skill', :url => {
:action => 'add_skill_form' }, :complete =>
visual_effect(:slide_down, 'add_skill', :duration => 0.5) %>
```

The :complete => visual_effect part of the statement specifies to use the :slide_down effect on the form, so that clicking the "Add Skill" link causes the form to appear in the div below the link (:update => 'add_skill') with a nice sliding effect. The last piece to making this form appear on the "create new project" view was to write the "add_skill_form" function in the projects controller:

```
def add_skill_form
    if request.xml_http_request?
      render :partial => 'add_skill'
    else
      redirect_to :action => 'project'
    end
  end
```

This function instructs that the ‘add_skill’ partial should be rendered when it’s called – in this case, when the “Add Skill” link is clicked:

Add Skill Form Expanded:



The show/hide form functionality works well, but I ran into some problems with the submit button on the “add skill” form. I use the :action => 'update_project' statement in the “add_skill” form (full code above) to call the following method from the projects controller:

```
def update_project
      @skill = Skill.new(params[:skill])
     respond_to do |format|
             flash[:info] = 'Skill Added.'
             format.html { redirect_to(:action => "new") }
    end
  end
```

Basically the method should create a new skill and then redirect back to the “create new project” view. This method worked, then I must have changed something else in the code that broke it. It was never a perfect solution – it still requires reloading the whole “create new project” page to add a new skill to the project – but it worked reasonably well. A future iteration of the application would ideally allow the form to be submitted without reloading the whole page.

The JavaScript for the “add file” functionality works slightly differently. The “Add Skill” link has some nice visual effects for the appearing skill form, but I was slightly frustrated that there was no way to hide the form once it had been called. For the “add file” implementation, I decided to use an additional script in the header section of the project layout. This script calls two visual effects from the included jQuery libraries – blindup and blinddown:

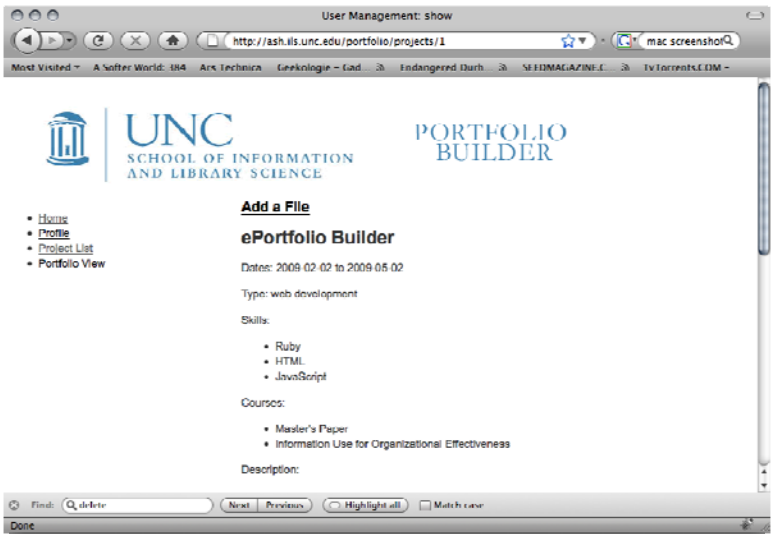
```
Event.observe(window, 'load', function() { init() });

function init() {
      $('addFile').observe('click',function() {
            if (!$('fileForm').visible()) {
                  new Effect.BlindDown($('fileForm'));
            }
            else {
                  new Effect.BlindUp($('fileForm'));
            }
      });
}
```

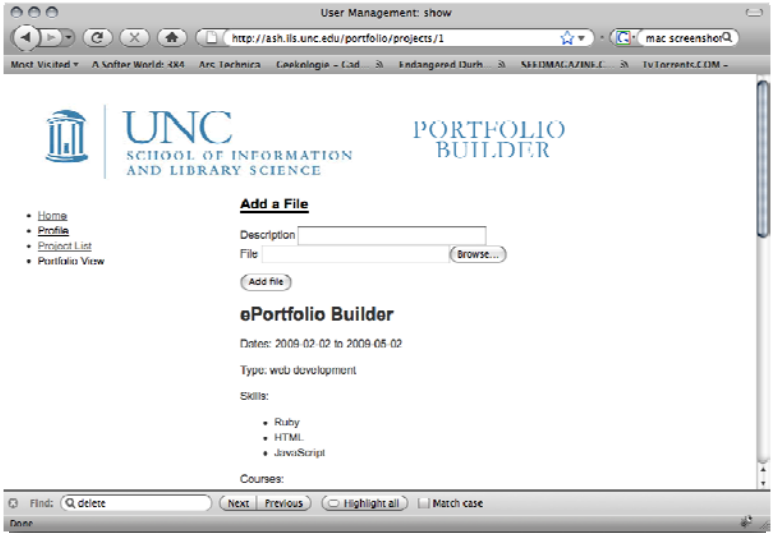
This script is embedded in the <head> section of the layout, so it’s loaded with the page. The script listens for click actions on the ‘addFile’ link – if the link is clicked and the fileForm div is NOT visible, the ‘fileForm’ div appears with a downward sliding effect. Otherwise, the div is hidden with an upward slide effect. I also embedded the form directly into the “show.html.erb” file for projects and set the div to hidden, rather than creating a partial view. This decision was partly due to the continuing problems I was

having with complex forms, and partly to give the user the option to hide the form if upon reflection no files were needed:

Add File Link:



Link Expanded:



The form for adding a file to project is a great success. The form appears and disappears without reloading the whole page, and when a file is added using the "Add File" button, the file is uploaded to the file system and appears in the "Files" section on the bottom of the "show project" view without reloading the page. Part of the ease of this implementation as opposed to the "Add Skill" implementation is the difference between associations – a file only belongs to a single project (one to many), while skills need to be available to every project (many to many).

However, I did run into some problems when trying to enable users to delete a file from a project. I modeled the "destroy" method on the basic "destroy" method that Rails includes in a generated controller:

```
def destroy
    @upload = Upload.find(params[:id])
    @upload.destroy

    respond_to do |format|
      format.html { redirect_to(@project) }
      format.xml  { head :ok }
    end
  end
```

Essentially this method should locate the file using the :id passed in though the params array, delete the file, and redirect the page back to the show project view. However, this method does not remove a file from a project – it deletes the entire project. Currently, the application cannot support this fairly basic action. More work will need to be done on this component in future iterations.

Redefining Association Types

Initially all many-to-many relationships in the application were implemented using the Rails has-and-belong-to-many (HABTM) association. The research and reading about many-to-many relationships that Erin, Kyle and I had done as of October, 2008 indicated that HABTM was the best (and only) way to implement this type of relationship in Rails. But when I began redesigning the presentation view, I ran into a bunch of problems with the projects-skills association. It didn't occur to me that there was any other way to define the many-to-many association until I was discussing my issues with a friend who's currently working as a Rails developer, S. M. Lunden (personal communication, April 10, 2009). She suggested implementing a has_many :through relationship instead of a HABTM.

The has_many :through relationship is new as of Rails 2+ and provides different functionality than a HABTM association. For my application, the most important functionality that this type of association provides is the rich association, meaning that it's possible to call an associated element and all its attributes through another associated element. This is not easy using a HABTM association, especially since the "push_with_attributes" method has been deprecated. Additionally, the HABTM

association itself has been unofficially deprecated, as many Rails developers prefer to use the rich association of the has_many :through relationship.

On a practical level, the differences between the two associations are that a HABTM relationship does not use a join model (only a join table) while a has_many :through relationship uses both a join model **and** a join table, but the naming conventions for each kind of join table are different. In order to implement the has_many :through association for projects and skills, I edited the project model by first deleting the "has_and_belongs_to_many :skills" statement. Then I needed to rename my join table from "projects_skills" (the HABTM naming convention) to a single word that described the relationship between the entities and was able to be pluralized (has_many :through naming convention). Although no single word came immediately to mind, I discovered that making up words is perfectly acceptable as long as they are meaningful within the application. I settled on "skillidentifications" as the title for my join table, as a project needs to identify associated skills and this would be the primary way the information is accessed within the application.

I created a migration to rename the table then entered the following in the projects model:

```
has_many :skillidentifications
has_many :skills, :through => :skillidentifications
```

The first has_many statement states a given project will have many entries in the :skillidentifications join table. The second statement expresses the association with :skills and explains that this association will happen through the :skillidentifications table. Similar statements replace the HABTM declaration in the skills table:

```
has_many :skillidentifications, :dependent => :destroy
has_many :projects, :through => :skillidentifications
```

Then I created the model for the skillidentifications table, where the belongs_to statements created the one-way association from the join records back to the parent tables:

```
class Skillidentification < ActiveRecord::Base
 belongs_to :project
 belongs_to :skill
end
```

Now I was working with a has_many :through association between skills and projects instead of a HABTM association, a change that made the presentation view redesign much easier to implement.

Presentation View Redesign

When I began solo development on this application, the interface for the presentation view consisted of a header indicating the name of the owner of the portfolio and a main content box with tabs located horizontally across the top. The tabs were automatically populated from the “Project Type” field in the project table, with the idea that although a project might exemplify many skills, it could only be described as a single type of project. Ideally, the “project type” would be a controlled vocabulary including types like: web development, databases, cataloging, etc. Creating a useful architecture for this kind of information fell outside of the scope of my project, and I deprecated usage of “project type” in favor of skills with the idea that a potential employer would be most interested in specific skills. Given this decision about the underlying data, I needed to redesign the presentation view so that projects were categorized by skill instead of type.

While the tabs in the original application were visually appealing, it was not possible to view the projects in two categories at once. I moved from a horizontal tab interface to a vertical expanding menu. This interface allowed the titles of projects in

multiple categories to be displayed concurrently, and also freed up the majority of the central area of the page to show details of a given project. I found a great jQuery menu on Marco Van Hylckama Vlieg's weblog "The Net is Dead" (2008). I saved the scripts into the application's public/scripts folder, and re-coded the presentation view's HTML appropriately. The menu immediately worked well with static data, requiring no additional configuration or troubleshooting.

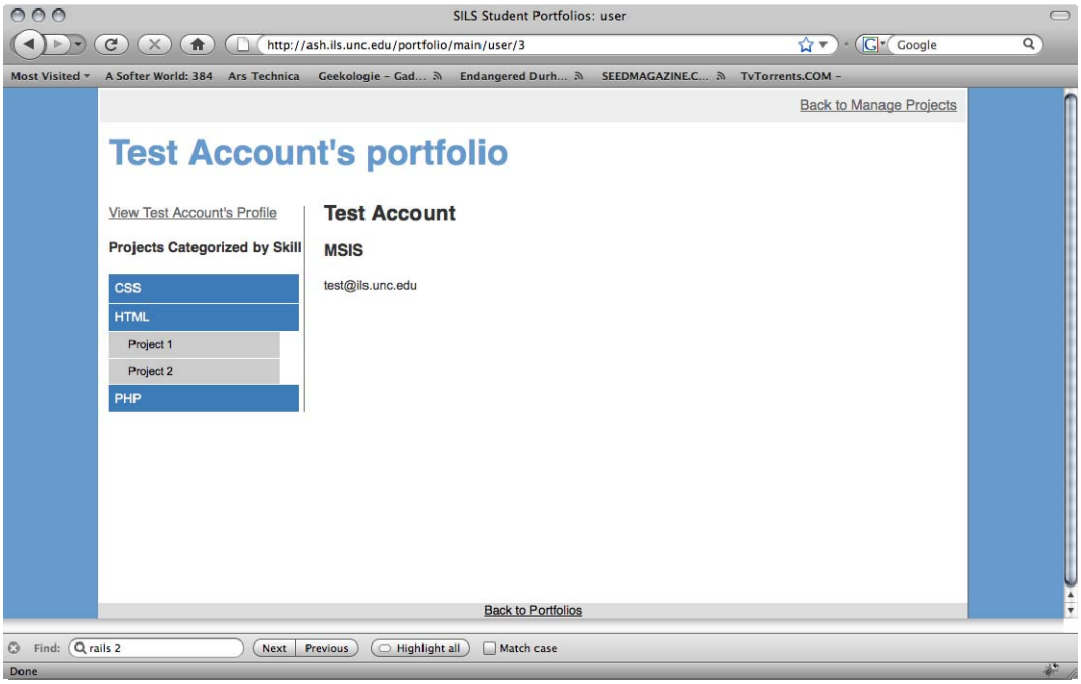
Populating the levels of the menu dynamically from a student's portfolio metadata proved to be slightly more difficult, until I implemented the has_many :through relationship between skills and projects. This association provided the rich functionality required to access all the attributes of projects through the skills they exemplify. For example, I was now able to access all the projects (and the attributes of those projects) associated with a given skill with by calling @skill.projects. This code exists in the "main" view ("main" is how the presentation view is referenced within the application) user.html.erb for a given user:

```
<% for skill in @skills %>

<ul id="menu3" class="menu noaccordion">

<li>
<a href="#"><%= skill.skillName %></a>
<ul>
      <% for p in skill.projects %>
      <%if p.user.id == @user.id %>
      <li><%= link_to_remote p.projectName, :update =>
      'project_view', :url => { :action => 'showProject', :id =>
      p.id }, :complete => visual_effect(:appear, 'project_view',
      :duration => 0.8) %></li>
<% end %>
<% end %>
</ul>
</li>
</ul>
<% end %>
```

The code builds a menu structured by skill name and projects for each skill nested inside the skill list. The rendered menu view appears on the left-hand side of the screen:

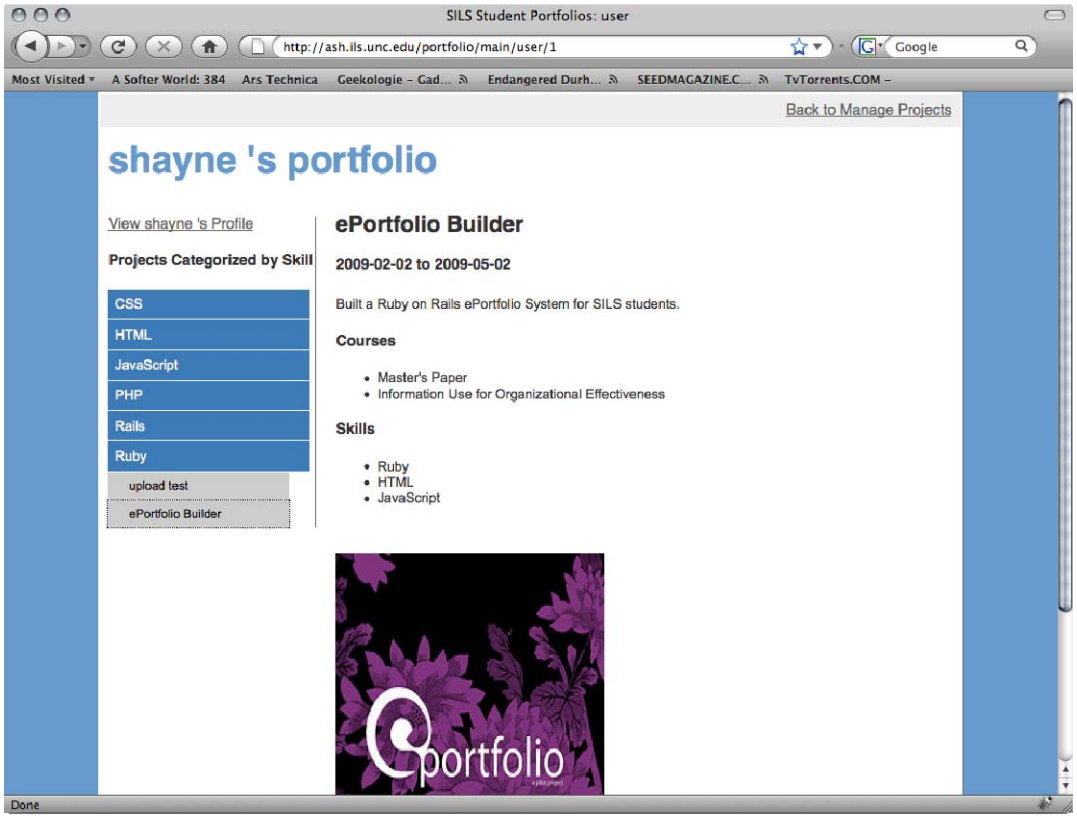


Skills and projects are both accessible through the following statements in the user method of the "main" controller:

```
@projects = Project.find(:all, :conditions => ["user_id = ?",
      params[:id]])
@skills = Skill.find(:all, :include => :projects,:group =>
      "skillName")
```

The :conditions specified for @projects variable make sure that only projects belonging to a certain user are returned. The :group statement in the @skills variable specify the proper alphabetic ordering of the skill names.

I used a the same setup as I did for "Add Skills" (described above) to populate the project view by clicking on the project's name in the menu list. The link_to_remote method updates the project_view div in the view with the _showProject.html.erb partial view, so that clicking on a project name in the menu list causes the project to be displayed in the space to the right of the project:



Using the AJAX link_to_remote method allows the show_project div to populate without reloading the whole page, a nice usability and speed feature for the user interface.

The presentation view redesign was extremely successful. The layout is improved by making multiple categories (skill groups) viewable concurrently for improved readability of the portfolio. The functionality is improved by categorizing projects by skill rather than by the not-fully-fleshed out “project type” attribute, and separating the page into clearly defined sections for menu and project view.

Future Directions

As my development was constrained by time and resources, I have many ideas for additional components and system functionality that I was unable to implement during this development cycle. In addition to refining aspects of components I've already discussed, the following components and elements would make the system more robust and complete:

- administrative dashboard:
    - provide students with the ability to show or hide their personal information in the portfolio presentation view
    - provide users with the ability to show or hide specific projects in the portfolio presentation view
- presentation view:
    - different organizing factors for projects, selected by end user (ex. courses in addition to skills)
        - develop controlled vocabulary for project types; make this an optional organizing factor
    - different style sheets for users to pick from; allow for greater individual expression across student users, rather than forcing every presentation view to have the same look
        - option to upload user-created style sheets
- integrated logon functionality: instead of forcing student users to register, hook the portfolio system into the UNC ONYEN authentication system
- extend the application to allow for grading/commenting from professors

Ultimately, an e-portfolio application must address the needs of the students, faculty and administration of the school. As such, the most important future direction of the application is to ascertain how it fits into the current curriculum and final examination structure. Testing the application in its current state with different user groups could result in the most useful information for continuing development.

Conclusions

The development of the e-portfolio web application was successful in that I achieved most of the goals I set at the beginning of development. Although this fairly alpha version of the system is not production ready, it's at an excellent starting point for another developer. Much of the organization of the codebase follows the standard Rails infrastructure – a developer familiar with the basic system design and MVC architecture should be able to continue working on the system without much preparation.

Ruby on Rails was a powerful choice for this application, as it provided so much of the basic functionality for free (without any coding). For instance, the create/update/delete functions for each object are built-in Rails methods, mitigating the tedium of re-coding each function for each new element in the application. Additionally, writing the application in this framework allowed me to gain knowledge of MVC architecture, the Ruby language, and experience in a different kind of web development than I was used to (ex. PHP/MySQL/HTML/CSS).

However, there are also many drawbacks to the Rails framework. It's very easy to quickly get an application up and running without any knowledge of the system (and even with very little knowledge of the Ruby language), but once more advanced configuration is required of a Rails application the configuration becomes more difficult

and less well documented. I found it to be a very steep learning curve once past basic structures and functionality. There are many excellent resources available to help learn Rails, many of which I used on a daily basis (see Appendix). However, Rails is still a relatively new technology, so searching for information for specific problems can be time-consuming and may not result in specific answers. I spent much of my time in "development" simply looking for answers to questions I had, often without much to show for it.

Ultimately, developing this application was a very rewarding experience, and I believe the work I've completed is an excellent start to a fully functional e-portfolio web application for SILS students.

References

Bates, R. (2008). Creating a weblog in 15 minutes with Rails 2. Retrieved March 2, 2009 from Ruby on Rails: Screencasts Website: http://rubyonrails.org/screencasts

Bates, R. (2007). Complex Forms Part 1, 2, and 3. Retrieved February 12, 2009 from Railscasts: Free Ruby on Rails Screencasts Website: http://railscasts.com/

Bates, R. (2009). Sortable Lists. Retrieved April 8, 2009 from Railscasts: Free Ruby on Rails Screencasts Website: http://railscasts.com/

Bhattacharya, M et al (2006). Work In Progress: E-portfolios in Computer Science and Engineering Education. *36th ASEE/IEEE Frontiers in Education Conference*, Session T1J.

Fitch, D. et al (2008). The Use of ePortfolios in Evaluating the Curriculum and Student Learning. *Journal of Social Work Education* 44 no3 37-54.

Hancock, D.R. (2007). Effects of performance assessment on the achievement and motivation of graduate students. *Active Learning in Higher Education*, v8 n3 p219-231.

Nicklas, J. (2009). UploadColumn. Retrieved January 28, 2009 from UploadColumn Website: http://uploadcolumn.rubyforge.org/

Ritzhaupt, A.D. & Singh, O. (2006). Student perspectives of ePortfolios in computing education. *ACM: Proceedings of the 44th annual Southeast regional conference*, 152-157.

Thomas, D. & Hansson, D. H. (2009). *Agile web development with rails (2nd edition).* Raleigh, NC: The Pragmatic Bookshelf.

Vlieg, M. (2008). Simple jQuery Accordion Menu – Redux. Retrieved March 23, 2009 from The Net is Dead Weblog: http://www.i-marco.nl/weblog/

Appendix

Rails Resources

Bates, R. Railscasts: Free Ruby on Rails Screencasts Website: http://railscasts.com/

Daigle, R. Ryan's Scraps Weblog: http://ryandaigle.com/

Raymond, S. (2007). *Ajax on rails.* Sebastopol, CA: O'Reilly.

Ruby on Rails Website: http://www.rubyonrails.com/

Ruby on Rails for Developers: http://dev.rubyonrails.com/

Thomas, D. & Hansson, D. H. (2009). *Agile web development with rails (2nd edition).* Raleigh, NC: The Pragmatic Bookshelf.