ProjectNomNom Final Report

CSE 454: Advanced Internet and Web Services
Autumn 2010

Noé Khalfa · Roy McElmurry · Josh Mottaz · Aryan Naraghi · Ryan Oman

Introduction

The Internet is a powerful resource for cooking, with a massive amount of recipes stored on many different websites. The number of online grocers, particularly ones who allow customers to order fresh ingredients such as produce and meat, has increased in recent years too. ProjectNomNom is intended to bridge the gap between the two complimentary services, offering a way to browse recipes and order their ingredients from Amazon Fresh (an online grocer, currently local to the Seattle area) with just a few clicks.

Project Goals

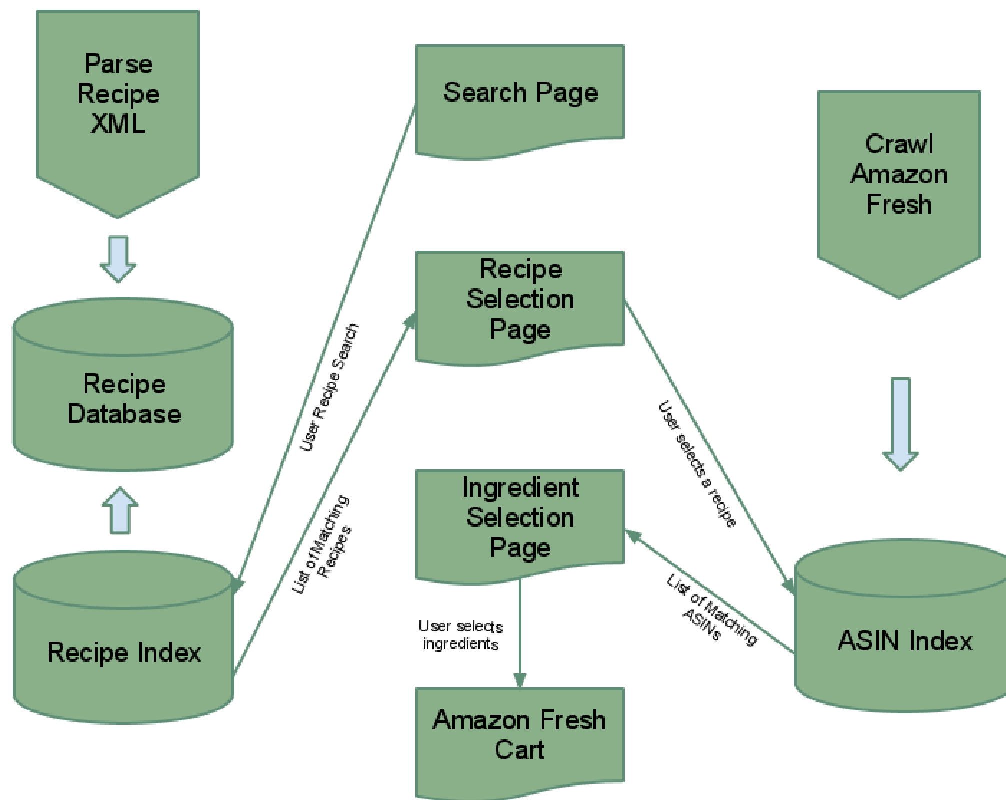
The project's primary goals are as follows: first, there must be some sort of search mechanism with a large number of recipes. Once the user picks a recipe, the Amazon Fresh index should match the ingredients with relatively high precision to items on Amazon Fresh. After the user confirms our matches the Amazon Fresh index made, they should be able to go to Amazon Fresh and checkout with their automatically filled cart. We were not able to automatically fill an Amazon Fresh cart because they do not have a public facing API; however, we did implement a user friendly way for users to manually add items to their cart.

For the recipe side, there were a number of smaller goals as well. The recipe index needed to be expandable so that, in the future, any number of other recipe websites could be crawled and archived for inclusion. Also, the system needed to be able to handle recipes with different measurement systems and account for those.

On the ingredients and cart population side, the goal was for the user to choose a recipe, and have the system select the product from Amazon Fresh that best fit the ingredient and quantity. However, we also wanted users to be able to modifying that, should the user want to pick a different product instead (if the match wasn't great or a larger quantity or different brand was desired).

Project Design

ProjectNomNom is made up of 5 main components, which can be divided into 3 layers. The first layer is the front-end/user interface layer, which is composed of the Rails view and controller layers. The second layer is the back-end, and contains the the Amazon Fresh Solr index and the combination of the MySQL database and the Solr index for the recipes, which is held together by the acts_as_solr_reloaded plugin and model layer of Rails. The third layer is the web-crawler layer, and contains the Amazon Fresh web-crawler and the code that parses both the Amazon Fresh data and the Recipe XML dump data.



Front End Layer

The front-end contains mostly Ruby, HTML, CSS, and JavaScript code. This section covers the view and controller layers in our Rails architecture. In the view layer we used the Rails ERB templating system to build the HTML pages. We also used the asset_packager plugin to deliver compiled JavaScript and CSS in production. The user authentication front-end was implemented with the formatter and validation plugins.

The features we included in designing the front end are as follows:

- Links to the origin of the recipe (if it was not submitted).

- Auto-complete on the search to minimize typing and searches needed.
- Amazon Fresh sign in.
- “Get Ingredients” page that allows users to select an ingredient from a list of matches, and allows user to add item to Amazon Fresh cart.
- Submit page that lets users submit recipes of their own, adding them to the database.
- Used Authlogic and Jquery plugins to provide a secure sign in.
- Minimal off site navigation through in-window pop-ups and JavaScript iframes.

The controller layer handles the communication between the view layer and the model and the Solr indices. The controller layer uses the Ruby rSolr gem to communicate with the Amazon Fresh index and retrieve ingredient matches for a given recipe. This takes us to the next layer in ProjectNomNom, the back-end.

Back-End Layer

The back-end layer is composed of: the Rails model layer, which is comprised of a MySQL database and the acts_as_solr_reloaded plugin; the Recipe Solr index; and the Amazon Fresh Solr index.

The Rails model stores all of our recipe data. We chose to store it in the database instead of the Solr index because the acts_as_solr_reloaded plugin along with the Rails model made it simple to index new recipes and keep the current index up to date. This made it easier to keep local copies of the recipe index on each of our development environments, and easily update the index as the quarter went on.

The acts_as_solr_reloaded plugin allowed us to abstract away the Solr index completely. We only had to update a recipe in the database using the Rails model and the plugin would keep the Solr index up to date. We did have to modify the acts_as_solr_reloaded plugin to provide better spelling suggestion results if no recipes were returned by a query.

For the Amazon Fresh index we chose to store the data in the Solr index instead of the database mainly because we wanted the searches to be very fast for the Ingredient matching, and having a MySQL database in between the view and the solr index slowed things down. To interact with the Solr index directly from the Rails controller layer, we used the Ruby rSolr gem, which abstracted away building an http request and made it more simple to communicate with the Amazon Fresh index.

Web Crawling and Data Extraction

One Solr instance indexes a database of recipes, while the other indexes a directory of crawled Amazon Fresh html pages. When the user searched for a recipe we query the Rails model, which returns us rows from the database. In the database, each recipe contains a list of ingredients that we use to query the Amazon Fresh index for likely matches.

To seed our recipe database, we used an XML dump from wikia.com. Unfortunately the data needed extensive cleaning before it was usable. We chose to use regular expressions to pull what we determined to be important information from the XML. With a very intricate regular expression, we were able to extract over 8000 recipes that had well-formatted ingredient lists.

We decided to not be as picky about the recipe descriptions or directions because they did not play a pivotal role in the system's functionality.

Two types of regular expressions were used to parse the ingredients from the recipe dump, those that included ingredient quantities and those that did not. If the regular expression with quantities did not match the ingredient text then we tried the regular expression without quantities. If both failed then we gave up and tossed the recipe out. In addition to these regular expressions it was necessary to replace many irregularities, such as rogue line break tags, with standardized strings.

We used Heritrix to crawl Amazon Fresh and index their catalogue. The crawled pages were placed into archival files, which were then were run through a Java parsing script and finally indexed by Solr. When setting up the crawler, we narrowed down the pages that were crawled to only search pages (including category indices) and product pages within the fresh.amazon.com subdomain. When examining URLs to determine whether we wanted to ingest a particular page, regular expressions were set up to normalize URLs with different categories, session IDs, and other attributes where the core product ASIN was the same. For category and search pages, URLs were standardized as well to account for page ordering and number of results to avoid duplicates.

The crawl netted roughly 90,000 pages. These pages were outputted from Heritix in 27 ARC files (commonly known as the Archive File Format) each about 95 megabytes in size for a total of two and a half gigabytes worth of Amazon Fresh data. The processing of this data was performed in Java. All pertinent files for this processing can be found in *nomnom/lib/amazon_fresh_backend_workspace/AmazonFreshBackend/src*.

The first step was in this process was extracting from the archive files. CrawledDataExpander is responsible for the unarchiving. The results of the unarchiving were HTML pages that were saved using their crawl timestamp. As part of the unarchiving process, the URLs of the pages were checked to make sure that each page was indeed a product page (this is something that was done during the crawling as well).

Next, the HTML pages were scraped for useful attributes to be placed in the Solr index. ProductInfoExtractor took the HTML pages and, using regular expressions, extracted the product names, prices, and all of the categories that the product belonged to (also known as similar items). The extracted values were used to create SolrInputDocument objects for each item that was crawled from Amazon Fresh. A SolrInputDocument is the object that Solr's Java API uses for indexing an item--it stores mappings of attributes to values.

Because of the large number of products we had to deal with, the SolrInputDocuments were not immediately indexed. Instead they were serialized to allow for more control over the indexing process and to maintain a persistent copy of the scraped data for later analysis (more on this when we discuss the removal of inedible items).

The Solr index was set up to store the following attributes:

- *ASIN*, the Amazon Standard Identification Number used to uniquely identify each item;
- *simpleProductName*, the product's simple title (the part of the title before the first comma);
- *additionalProductInfo*, the product's extended title (the part of the title after the first comma which includes a more detailed explanation of the product);

- *priceInCents*, the price of the product in cents; and
- *similarItems*, all of the categories associated with a given product (also known as browse nodes).

The preceding attributes are defined in the Solr index's schema file which is located at *amazon-fresh-solr/solr/conf/schema.xml*. The attribute *similarItems* is multivalued because most products belong to three to five different categories. Once the Solr server was started, the serialized SolrInputDocuments were unserialized and indexed in batches using ProductIndexer. In all, 61,186 products were entered into the Solr index (the drop from 90,000 original documents resulted from non-product pages and duplicate pages being eliminated).

Initially, the product matches for items like flour and salt were very poor. To address this, the searching on the Solr index was extended to the similarItems attribute to allow more general terms like salt and flour to help return relevant results. This greatly improved the mapping of ingredients found in recipes to items in Amazon Fresh's catalogue.

Unfortunately, a lot of the ingredients ended up mapping to inedible items on Amazon Fresh. For example, the ingredient "turkey" mapped to a book on Amazon Fresh called *10 Fat Turkeys*! To prevent this from happening, we decided to remove inedible categories (from the similarItems attribute) from the Solr index. This is where having the SolrInputDocuments serialized helped.

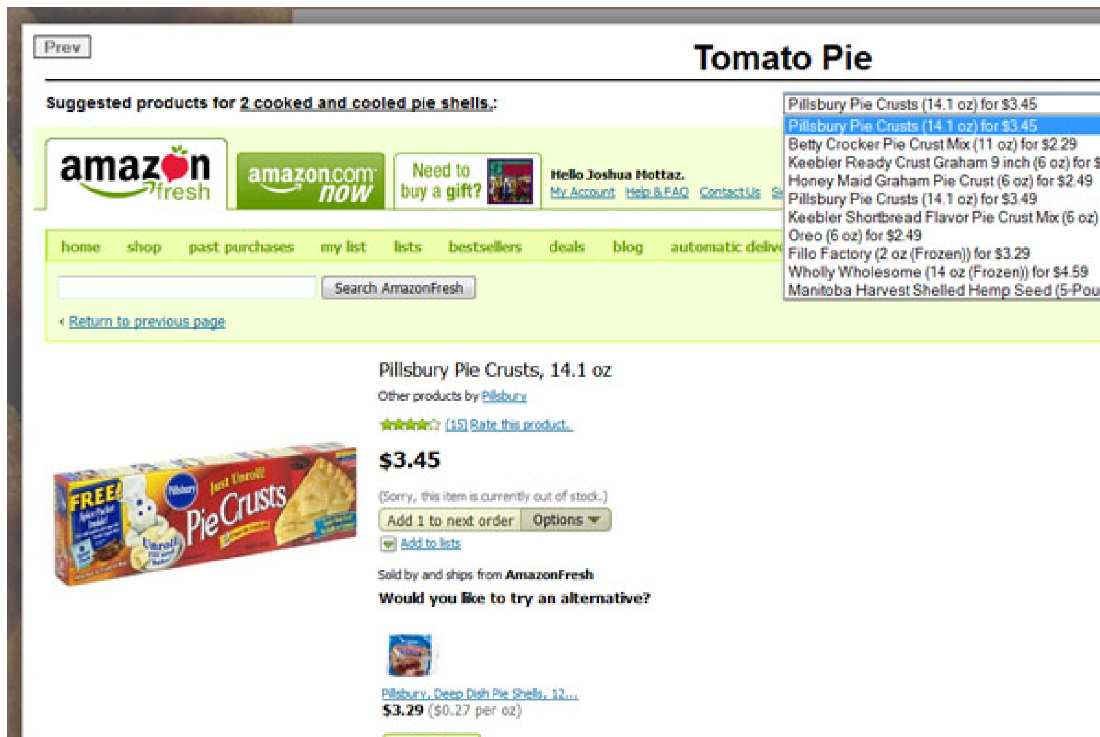
The SolrInputDocuments were deserialized and classified to a set of all similarItems attributes (or categories) that existed in the hopes of removing inedible categories. Unfortunately, this yielded 9,930 unique categories. To address this issue, we decided to map each category to its frequency and remove the most-frequently occurring inedible categories from the index. This made a lot of sense given the fact that most items had three to five categories.

CategoryExtractor was responsible for creating a map of categories to counts and CategoryRemoverClient was a client program that took similarItem values and performed deletes on the Solr index for the given values. After going through only about 150 of the top categories, we were able to shrink the size of the Solr index from 61,186 Amazon Fresh items to 32,988. This pruning significantly improved our ingredient to Amazon Fresh product mappings.

User Flow & Usage

1. Home page, from here the primary focus is the search bar, which the user can use to find recipes. This page also shows the search suggestion functionality.

2. Once the user has entered a search query, a results list is displayed where they can preview recipes by expanding them, or click the “Get Ingredients” link to open the Amazon Fresh cart loader window.



3. At Amazon Fresh, the user is prompted to log in and push the “I Have Signed In” button when finished. From here, the user is presented with a drop down menu with the ingredients from the recipe, matched to Amazon Fresh. The user can select different ingredients from the menu, resulting in a refresh of the Amazon Fresh iframe. When satisfied with the ingredient, the user clicks the “add to cart” button, and then hits the next button to select the next ingredient.

Experimentation & Testing

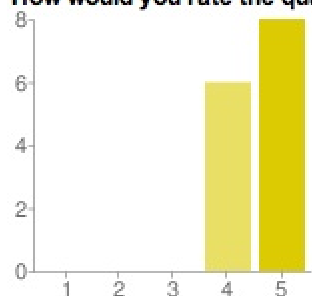
Our project is meant to be usable by a wide range of people with varying degrees of technical proficiency, so user testing and feedback is essential. To get feedback, we created a rubric allowing users to grade the search page, search results, integration with Amazon Fresh, result effectiveness (how close the items matched what a reasonable user would expect if they were trying to find an ingredient themselves), and the site as a whole.

For the search, results and Amazon Fresh integration pages, we asked the user to rate the visual appeal, ease of use, and speed (how fast the page was served) on a 1 - 5 scale. We also asked the user to rate the effectiveness of both the recipe search results and the ingredients matching. In addition to the ratings, each category had a comment box at the bottom for an potential errors or accolades the user wanted to mention. At the end of the form, a general comment box provided an area for any extra input the user wanted to mention.

The goal was to collect feedback, then examine it and look for potential areas users have identified that consistently are problematic. We gathered some useful data from the user feedback. One small thing that bugged people with larger screens was that the background image tiled instead instead of stretched to fit their screens. We also learned of several usability fixes that could be made. Some people mentioned that a little more direction would be helpful as one navigates the website. Most importantly people noticed as we did that there is room for improvement on the quality of ingredient matching with Amazon Fresh. If we were to continue

with the project I think we would need to respond to all of this feedback. In general though, the users seemed to like our product.

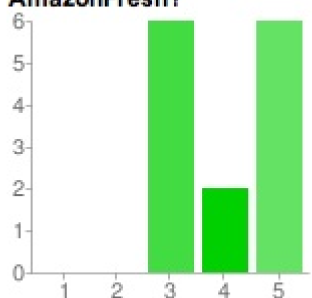
How would you rate the quality of the recipe search results?



Worst Best

1 - Worst	**0**	0%
2	**0**	0%
3	**0**	0%
4	**6**	40%
5 - Best	**8**	53%

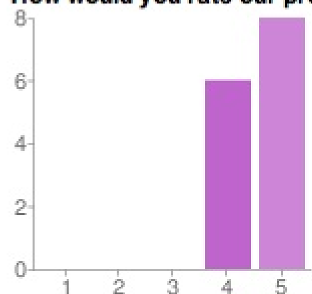
How would you rate the quality of the ingredients from the recipes as they were matched to AmazonFresh?



Worst Best

1 - Worst	**0**	0%
2	**0**	0%
3	**6**	40%
4	**2**	13%
5 - Best	**6**	40%

How would you rate our project overall?



Worst Best

1 - Worst	**0**	0%
2	**0**	0%
3	**0**	0%
4	**6**	40%
5 - Best	**8**	53%

In addition to peer evaluation we also performed some self evaluation. Before we began the quarter we designated several recipes as evaluation targets. Three main metrics were used to evaluate the effectiveness of our website; the page rank of the first relevant recipe, the number of ingredient parsing errors present in that recipe and lastly the number of top ranked ingredient errors for that recipe. From this data we discovered that when the recipe search term does return results it generally has the desired recipe as the first of second hit. We only have about 8,800 recipes in our database, so we are quite confident that with more recipes we would be returning meaningful results for all of the search terms. Parsing the ingredients also proved to work very well. When we get to the quality of Amazon Fresh ingredient matching our results are not as good. In general about ¼ of the ingredients give results that are not the greatest. They usually are not wrong, just not right. For instance, salt generally results in a $20 container

of black truffle salt.

So, in summary, our testing showed that the product has come along nicely and is definitely a working system. However, if we were to continue to work on this project we would need to spend time improving some UI specifics as well as the effectiveness of our ingredient matching system.

Group Dynamic

From the beginning we drew up a list of tasks and began to split them up based on who was interested in which task and who had skills in which areas. However, in order to keep people happy and productive we made sure to not assign strict boundaries. By the end of the project everyone had spread their time and learning to different areas and got a feel for subjects they didn't know, as well as a stronger look into fields in which they had already worked. The success of this project really had to do with a solid group dynamic and making sure control was shared by all team members, while also having individual responsibilities laid out.

On the Horizon

There are a number of features we want to add and expand upon. The primary one we had intended to implement was auto cart loading. Early in the project, we contacted Amazon and they mentioned the possibility of adding an API allowing an external service (us) to populate a cart with a list of items. Unfortunately, this didn't materialize, and we were forced to implement a manual, item by item solution instead. Given more time, we want to implement an item confirmation, then add the ingredients in just one click.

We also intend to expand user account support, currently users can create accounts and log in, but there are no additional features logged in users receive. Our goal is to implement features oriented around these accounts, such as favorite recipes and saved carts. We also would like to tune our ingredient matches based on a user's previous choices.

Another feature, indirectly implemented, is the ability to order ingredients for multiple recipes. Though a user can add a recipe's worth of items to the cart, find another recipe, and do the same thing, it might be helpful to allow the user to save recipes on our site. The major benefits would be twofold: first, as the user browses he/she can keep track and easily add or remove recipes he/she would like to order, then confirm and all the ingredients at once. Second, recipes with overlapping items can have those consolidated into one, which avoids purchasing unnecessary duplicates of ingredients.

Conclusions

We feel that we have accomplished and learned much this quarter during this project. Only one group member knew Ruby on Rails and Git when the quarter started, and now most of the group members have a working knowledge of both. Several of us also have become competent when dealing with search in Solr and web-crawling with Heritrix. Overall, the project appears to be well-received, and we feel ProjectNomNom has much potential as tool for those

looking to shop and cook from the convenience of their homes.

Appendices

- Division of Labor
  - Noé Khalfa
    - UI design in Ruby on Rails, html and javascript
    - Graphical and informational layout
    - User submitted recipes functionality
    - Help and About page
  - Roy McElmurry
    - Parsing the recipe XML dump
    - Ingredient matching page architecture
    - Auto-completer front end
    - Search term bug fixes and query escaping
  - Josh Mottaz
    - Setup and maintainance for Heritrix EC2 instance
    - Heritrix crawl setup and monitoring
    - Pre-Solr Amazon Fresh data filtering
  - Aryan Naraghi
    - Proposal write-up and page design.
    - Processing of crawled Amazon Fresh data which included the extraction of the archive files from Heritrix, sanity checks on the HTML files, and scraping of useful attributes.
    - Indexing of Amazon Fresh into Solr and the removal of inedible items from the index.
    - Recipe auto-suggest feature backend.
  - Ryan Oman
    - Amazon EC2 setup and maintenance
    - Rails app architecture, Solr integration with Rails
    - Main search results page
    - Ingredient selection/conflict page
    - Training the group in Ruby on Rails and on Git
    - User authentication front-end and back-end

- External Code Used
  - JavaScript
    - Jquery library - http://jquery.com/
    - Validation plugin - http://bassistance.de/jquery-plugins/jquery-plugin-validation/
    - Autocomplete plugin - http://docs.jquery.com/Plugins/autocomplete
    - FancyBox plugin - http://fancybox.net/
    - RandomBackground plugin - http://charles-harvey.co.uk/plugins/randomBackgroundImageChanger/

        - Template plugin - https://github.com/jquery/jquery-tmpl
    - Ruby
        - Ruby on Rails - http://rubyonrails.org/
        - Rails acts_as_solr_reloaded plugin - https://github.com/omanamos/acts_as_solr_reloaded
        - Ruby rsolr plugin - https://github.com/mwmitchell/rsolr
        - Rails authlogic plugin - https://github.com/binarylogic/authlogic
        - Rails asset_packager plugin - http://synthesis.sbecker.net/pages/asset_packager
    - Images
        - Banner images/cartoons from Genevieve Bienvenue and Gabe Groen
        - Background images from Flikr with Creative Commons license

- Usage Instructions
    - Website: www.projectnomnom.com
    - Browser requirements: Google Chrome (latest version), Firefox (latest version), Safari (latest version)
    - Usage: http://projectnomnom.com/help