Project Overview

The objective of this project is to create a centralized content repository for Dining and Lab locations on the RIT campus. A new content collection application was created for authorized members of the campus community. This new application provides a self-service method to collect public content used by various RIT web sites and RIT mobile applications. Public content includes descriptions, menus, open and closed hours, images, and geo location coordinates.

In addition to the admin collection tool, the project team developed an API to make the centrally collected data available to external systems, including but not limited to Drupal and non-Drupal Department Web Sites, RIT Maps and RIT Mobile.

Motivation

The current content collection landscape relies on several sources of information which are difficult to keep up-to-date and consistent. Descriptions and images of some locations have been found to be out of date, and hours of operation are sometimes listed incorrectly. The applications used to update this data are very old and hard to use. In particular, Dining Services employees have expressed frustration using the “Web Menus” tool, which is over a decade old.

Where CLiC fits in the ITS landscape. Map Admin Tool will be replaced in the future.
Scope

Project scope includes:
● Responsive web application which provides the following features
● Managing basic information about a location, including description, website link, and image URL
● Managing hours of operation
● Managing menu items and menus for Dining Services locations
● Multiple user levels with different permissions and responsibilities
● Storage of all of the above content
● API for accessing all of the above content
● Import from existing location content database

Project scope does not include:
● Departments other than Dining and Labs
● Additional special content outside of hours and menus
● Dynamic way of adding special content types within the application
● Additional user levels, or a means of creating them within the application
● Integration with other systems

Basic Requirements

The basic requirements for this application have been broken up into 8 categories by functionality, and are described below:

C1  Home
● A user shall be able to log into the system.
  ○ This login is authenticated using shibboleth, and the user is authorized using the permissions they have previously been assigned.
● A user shall be able to view assigned locations.
  ○ After a user is logged in, data pertaining to the locations or departments they are assigned to is retrieved from the database and shown to them on the home page.

C2  Basic Content Addition
● A user shall be able to add an image for the location
● A user shall be able to add a description for a location
● A user shall be able to add a website link for a location

C3  Basic Content Editing
• A user shall be able to edit an image URL for a location
• A user shall be able to edit a description for a location
• A user shall be able to edit a website link for a location

C4 Basic Content Removal
• A user shall be able to remove an image URL for a location
• A user shall be able to remove search keyword for the location
• A user shall be able to remove a description for a location
• A user shall be able to remove a website link for a location

C5 Special Content Management
• A user shall be able to edit open/close hours for a location
• A user shall be able to edit menu for a location
• A user shall be able to add a menu item
• A user shall be able to add a meal name to menu item
• A user shall be able to add description to menu item
• A user shall be able to add nutrition facts to menu item
  ○ This specific requirement was deemed out of scope for the first version of this application.
• A user shall be able to add a price to menu item
• A user shall be able to set a date for menu item
• A user shall be able to edit a menu item
• A user shall be able to delete a menu item
• A user shall be able to create menu group

C6 Staff Management
• A user shall be able to add a location staff to a location
• A user shall be able to remove a location staff to a location
• A user shall be able to view a report of staff by location
• A user shall be able to add a location manager to a location
• A user shall be able to remove a location manager from a location
• A user shall be able to add a location staff to a location
• A user shall be able to remove a location staff from a location
• A user shall be able to view a report of staff by location
• A user shall be able to add a department manager to a department
• A user shall be able to remove a department manager from a department

C7 Location Management
• A user shall be able to add a location to the department
• A user shall be able to remove a location from the department
• A user shall be able to edit the building and room of a location

C8 Log Keeping
• The system shall store a report of staff by location, which will be accessible by managers.
• The system shall keep track of application activity (changes) by department, which will be accessible by managers.

Constraints

The Enterprise Web Application (EWA) team within ITS maintains many projects like this, so there are certain technologies that they are familiar with or use as a standard. We were required to use an Oracle database for the content storage. The data we stored had to cover the data in the systems we were to replace; for example, we had to include an “mdo” field for locations to link to map data. We were given lists of technologies to choose from for different parts of the project, with the option of choosing different ones if we could convince the team why they would be better choices and make sure they would be able to maintain it when we were finished.

The EWA team strongly suggested that we develop two separate projects: one for the web application and one for the API. We were provided with one server for development, and partway through the year ITS set up two test servers. Other resources provided to us include RIT’s web standards, several useful web development tools, demonstrations of the current systems, and some feedback from Dining Services staff.

Development Process
We elected to follow Feature-Driven Development. It is one of the members of the agile development family with a focus on working code over large amounts of documentation. This process involves laying out the model of the system through high level walkthroughs followed by more detailed discussions. Once the model is laid out a feature list is built out of the expected requirements and discussions with the stakeholders. Following this, the features are prioritized and scheduled, and development begins. The features were broken up into classes which are two week long sprints after which the current version of the features were reviewed with the customers to receive feedback. This allowed for any changes to be made earlier in the development process, and it prevented larger setbacks.

We used Trello to track our tasks, and a spreadsheet document to track the estimated times as well as actual times for each task.

A template outlining the main events and tasks within each sprint is shown below.

<table>
<thead>
<tr>
<th>Design Week</th>
<th>Thursday</th>
<th>Saturday</th>
<th>Sunday/Monday</th>
<th>Tuesday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Week</td>
<td>Team Meeting -review feedback -requirements -test plan -discuss design</td>
<td>Team Meeting -optional/virtual -review design -prep for Sponsor Meeting</td>
<td>-act on feedback -start development -email agenda</td>
<td>Sponsor Meeting -optional -discuss design -review doc updates</td>
</tr>
<tr>
<td>Build Week</td>
<td>Team Meeting -development coordination</td>
<td>Team Meeting -optional/virtual -review feature status -prep for Sponsor Meeting</td>
<td>-test -deploy -email agenda</td>
<td>Sponsor Meeting -discuss feature implementation -gather feedback -refine requirements for next feature</td>
</tr>
</tbody>
</table>
Project Schedule: Planned and Actual

At first, we scheduled several weeks of ramp-up time and set a date when we’d start coding so ITS could get a dev server and database ready. During that time, we performed the initial stages of the feature-driven development process, including creating overall models and organizing feature classes.

In the feature organization phase, we estimated how much time it would take to complete each feature, in weeks of development time. Some features were relatively small and would take less than one week, while others we expected to take more than one. We prioritized this list with dependencies in mind, shown below, and then grouped them into timeboxed iterations to form the schedule for the rest of the project. Each iteration consisted of one week of design and one week of development. These time-boxed iterations helped us stay productive and make sure we kept a steady pace.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Feature Group</th>
<th>Dependencies</th>
<th>Estimation*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C1 (login, view locations)</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>C2, C3, C4 (manage basic content)</td>
<td>C1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>C5 (hours)</td>
<td>C1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>C5 (menu)</td>
<td>C1</td>
<td>2-3</td>
</tr>
<tr>
<td>5</td>
<td>C6 (user levels)</td>
<td>C1</td>
<td>.5</td>
</tr>
<tr>
<td>6</td>
<td>C6 (manage location staff)</td>
<td>C1, C6 (user levels)</td>
<td>.5</td>
</tr>
<tr>
<td>7</td>
<td>C6 (svc desk management)</td>
<td>C1, C6 (user levels)</td>
<td>.5</td>
</tr>
<tr>
<td>8</td>
<td>C7 (manage locations)</td>
<td>C1 C6 (user levels)</td>
<td>.5</td>
</tr>
<tr>
<td>9</td>
<td>Reporting</td>
<td>Logging from other features</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Additional scope</td>
<td>Product is “complete”</td>
<td>n/a</td>
</tr>
</tbody>
</table>

*Estimation: weeks of development time, not including design

Most of these feature groups were assigned one iteration. We dedicated one iteration for managing menu items plus two more for menus themselves (integration with hours), since that seemed to be the most complex part of the product both in data and in UI. The last few were
combined into one iteration since they were relatively simple once their dependencies were complete. We feel that we estimated the schedule well, since each iteration seemed to require roughly the same amount of work and we were able to reliably follow the schedule.

There was some technical debt buildup, mostly in the realm of testing, but for the most part we delivered features on-time. Partway through the second semester, we were asked to move the code-complete date forward a couple weeks. To accommodate this, we combined a few feature classes into one iteration and eliminated a couple minor ones.

**System Design**

Our system architecture was primarily driven by the specificity of our architecturally significant requirements, as well as suggestions made at the start of the project by the sponsors. They requested that the application be developed in two partitions: a CLiC App and a CLiC API.

![Overall Architecture, including both CLiC App and CLiC API projects](image)

The CLiC App serves as the responsive front-facing web application. It is the single, central point of data entry and will be used by the Staff and Managers at multiple departments and locations across campus. The App was built using Foundation and AngularJS, mobile first. What this ensured was that before the regular desktop-view of a page was created, its mobile accessibility and usability was taken into consideration. This design model also guaranteed seamless responsiveness of the application for big data entry at a laptop or desktop computer, as well as updates or quick changes on the go via mobile phones and tablets. Some of the other technologies used include HTML5, CSS3 for web page layouts.
The CLiC API on the other hand, serves as the backend for the CLiC App. It is used as the data store for all the information collected by the application. It currently only works with the CLiC App, but it is set to interface with other applications in the future - providing information to many other services in the RIT data network. The API is written in Java, and it works with the Spring Framework, as well as an Oracle Database.

To accommodate the requested responsive design, we decided to put most of the application logic on the client, which consists of Partials (Views), Controllers, and Services (Models). This Client will only directly interact with the CLiC App server - which consists of a Proxy API, an intermediary between the CLiC App Client and the CLiC API. We designed the system this way as a means to improve overall security in the application by reducing the number of vulnerabilities in its threat model. This Proxy API in the CLiC Server is written in Java as well, and is also authenticated by shibboleth as is consistently done across every other trust boundary in the application.

Even though we designed the system to require shibboleth authentication across multiple points of access, security attacks were not our primary concern. We decided that the information being stored on the application is not sensitive at all, and so there would not be much motivation for any targeted attacks on it. However, our security checks were put in place to prevent accidental breaches of access or elevations of privilege by unauthorized users.

The following diagram is a high-level design of the CLiC App portion. “CLiC App” consists of two javascript files, app.js (containing routing information) and ctrl.js (containing all controllers); “CLiC Svc” is a single javascript file svc.js (containing all services); and ngRoute is an extension of AngularJS.
CLiC is a single-page application, meaning the client downloads index.html and the javascript and css files it references, and the rest is done with javascript and HTML partials. The CLiC App component uses ngRoute to translate URL strings (after the “#”) and load the appropriate partial and controller. These follow the MVC pattern, using Angular “services” as models. The services send AJAX calls to the Proxy API to provide or submit requested data. The Proxy API makes sure the user is logged in, and then sends the user’s ID and an API key along with a copy of the request to the CLiC API (not shown).

**Process and Product Metrics**

To compute product metrics, the measurements we collected were the size of the project in number of lines of code, as well as the number of features. From these measurements, we created measures such as defects per 1000 lines of code (KLOC), defect removal effectiveness, percentage test coverage, and team effort per feature - in addition to time tracking. We also collected process metrics pertaining to requirements volatility - rate of requirements change. We definitely should have put more effort into gathering usability metrics. A SUS score would have given us more insight into how the customers really felt about the UI. However, for our project, we sent out an informal survey to the dining service staff - our final users - and received nothing but positive feedback about the ease of use and ease of learning our application.
Product State at Time of Delivery

The core in-scope features were completed. At the time of delivery, the product was running on ITS’s test servers and undergoing user acceptance testing. ITS plans to push it to production in June. Roughly half of the code and functionality is covered by automated unit and end-to-end tests, respectively, and there are a couple of known bugs that we weren’t able to fix in time. We provided a document to the EWA team listing these bugs and describing how to run the tests and maintain the software.

We negotiated some scope adjustments early on in the first semester. The original proposal described geolocation information and integration with other systems, but we agreed that these would be left to the EWA team after we finished the vital content and staff features. One feature that we failed to deliver is logging and reporting. We had to neglect this feature to accommodate moving the code-complete date forward a couple weeks, combined with resolving technical debt buildup.

Project Reflection

Our project was fortunate to have on-campus sponsors, so we were able to communicate with them fairly well. We met almost every week in a conference room and showed them newly-developed features in person. This facilitated our feature-based iterative process. We think that feature-driven development was the perfect fit for this project. We assumed that requirements would be unclear yet non-volatile because we weren’t working directly with the end users (dining services) but ITS had a clear plan for the project. The overall modelling phase of FDD helped us understand the basic requirements and helped the EWA team see how we understood them. We got into more specific requirements at the start of each iteration’s design stage as well as from feedback after the development stage.

One FDD practice we didn’t follow but feel we should have is assigning features to developers. Instead, we divided the team into different areas of the system (i.e. app, api), so each became an expert of that area. Unfortunately we did not do a good job of ramping up the other members into the different areas. Blockers occurred frequently, as one member waited for the expert in the other area to complete a task.

After the first semester, we decided that we wanted to adopt test-driven development. We had always outlined a test plan during the design phase, but the tests were only written at the last minute after the feature was implemented. We failed to do this in the second semester.
For the product, we were mostly satisfied with our technology choices. In the future we probably wouldn’t use Oracle, but we didn’t have that choice. Bootstrap might have been a slightly better CSS framework than Foundation since it has a more mature integration with AngularJS. In the future we would do more usability testing and adjust the UI accordingly (such as changing the button color).