# jobSalt

### Team Pepper

|  |  |  |
| --- | --- | --- |
| Ross Kahn |  | David Lamont |
| Daniel Wallach | Colton Presler | Chris Ray |

### Project Sponsors

### RIT Office of Cooperative Education & Career Services: Jim Bondi

### University of Arizona Office of Career Services: Christopher Jansman

### Faculty Coach

Daniel Krutz

### Project Overview

There may be tools provided by a University to help students find jobs, but often times those tools only present a small fraction of the available jobs for a given field. The tools are also impersonal, forcing students to apply to companies without any knowledge of people who may work at the companies; there may be alumni from the students' schools that would be willing to help the student apply. Furthermore, the tools often don’t provide any way to compare offers they may receive from companies, to salaries of other people who went to the same University for the same field of study. Lastly, students often need help finding housing near their job placement location, and current tools do not help them find or discuss that housing with other students who may have lived in the area.   
  
This project, named jobSalt, aims to solve all of these problems by:

1. providing to the user jobs found by the University, as well as jobs found on the Internet
2. allowing users to see any alumni who may currently work at the company of their choosing (if those alumni wish to be contacted)
3. providing salary data collected by the University for a given field of study
4. allowing students to discuss housing options based on a specified company or location.

jobSalt is implemented as a website, with a design that will make it easy to use on internet-enabled mobile devices. It uses the API’s of various data source websites to find job postings, alumni contact information, salary information, and housing reviews, and has an architecture that allows for the low-effort development of modules to pull data from new sources in the future. jobSalt is designed in a University-independent way, so that any school may use it after some configuration.

A high value is placed on making jobSalt usable and aesthetically pleasing, as well as functional.

### Basic Requirements

The jobSalt system has four core features, along with a variety of general requirements:

### Job Search Feature (High Priority)

The system must collect job posting information from various external sources and aggregate them into a list, which will then be shown to the user.

Each job posting must contain various pieces of information, which the user can filter by. These filters are, for the first release:

* Field of Study – the academic major that this job posting relates to
* Company – the name of the company for this job posting
* Job Title – the position title for this job posting
* Location – where the job is located (restricted to the United States)
* Source – which external data source did this job posting come from
* Keyword – searches all areas of the job posting

When the filter values are modified, a new request for data is sent by the system and fresh information is displayed, respective to the filters enabled.

The jobs posted must be in date order, from most recent to oldest.

The job posts are initially “collapsed” meaning only a portion of the information is displayed to the user. When the user clicks on a post, the post will “expand” showing the user all of the possible information from the post, along with several buttons to navigate the user to different parts of the site (or to the external source of the job posting).

### Alumni Connect Feature (High Priority)

The system shall collect alumni contact information from the University’s alumni database and/or any external sources, which will then be shown the user.

Each alumnus must contain various pieces of information, which the user can filter by. These filters are, for the first release:

* Field of study – the academic major(s) of the alumnus
* Company – the name of the company the alumnus is working for
* Location – the geographical location of the alumnus
* Keyword – searches all areas of the alumnus post

The alumni posts are initially “collapsed” meaning only company names are displayed to the user. When a user clicks on one of the company names, the post will “expand” showing the contact information for all of the alumni who work for that company.

### Salary Comparison Feature (Medium Priority)

The system shall collect salary information from the University’s salary database and/or any external sources, which will then be shown to the user. The salary information should be grouped by Field of Study. Each data source should be displayed as a separate graph.

Each graph of salary information shall contain the mathematical Average and Median of the set of salaries for a given Field of Study collected by the system. In addition, the graph should display at least two standard deviations. If no Field of Study is specified, the resulting graph(s) shall display information based off of all salaries independent of Field of Study.

A graph for a specific Field of Study can be shown to the user if the user types in that Field of Study in the search box provided. Each graph shall display the source of the information, as well as the currently applied Field of Study search. If there is no information for the Field of Study the user entered, no graph will be displayed.

### Location Review Feature (Low Priority)

The system shall collect housing reviews from jobSalt’s housing review database and/or any other external sources, which will then be shown to the user.

The Location Review feature is unique in the sense that users can input reviews, which the system will then store in a database on the jobSalt instance’s server. When a user inputs a review, they must specify:

* A star rating, corresponding to how much the user enjoyed their experience at a given location (1 star being the lowest, and 5 stars being the highest rating)
* A title for the review
* A description of the experience at a given location
* A location, which is specified by using the Google Maps auto-fill API

A system administrator can be specified in the configuration. System administrators can delete any post that was input into the system by a jobSalt user. Any user who input a post may delete their own posts at any time, but a normal user can not delete anyone else’s posts. Administrators may not delete posts that were brought into the system from an external source (in other words, a post that is not in the jobSalt housing review database).

Each housing review post must contain various pieces of information, which the user can filter by. These filters are, for the first release:

* Keyword – searches all parts of a review (see above) for a given string
* Location – brings up all reviews within a set radius of the location specified with the Google Maps auto-fill API

Housing posts are automatically sorted in descending star-rating order from the highest rated to the lowest rated. This is true regardless of the filters applied.

### General Requirements

A Twitter feed, connected to the University’s choice of Twitter accounts, must be shown on each normal page (but not on the configuration or log-in pages).

A user must log in with a valid University username and password (verified with LDAP or another authentication method).

The site must be configurable from a configuration page, accessible only to users whose usernames are contained in the system configuration file. On the configuration page, the following must be allowed to be configured:

* A .css file containing the aesthetic theme for the site
* Which features (between jobs, alumni, salary, and housing) to be enabled or disabled
* Whether the Twitter feed is enabled or disabled. Also, the necessary fields to initially configure the Twitter feed.

### Architectural Requirements

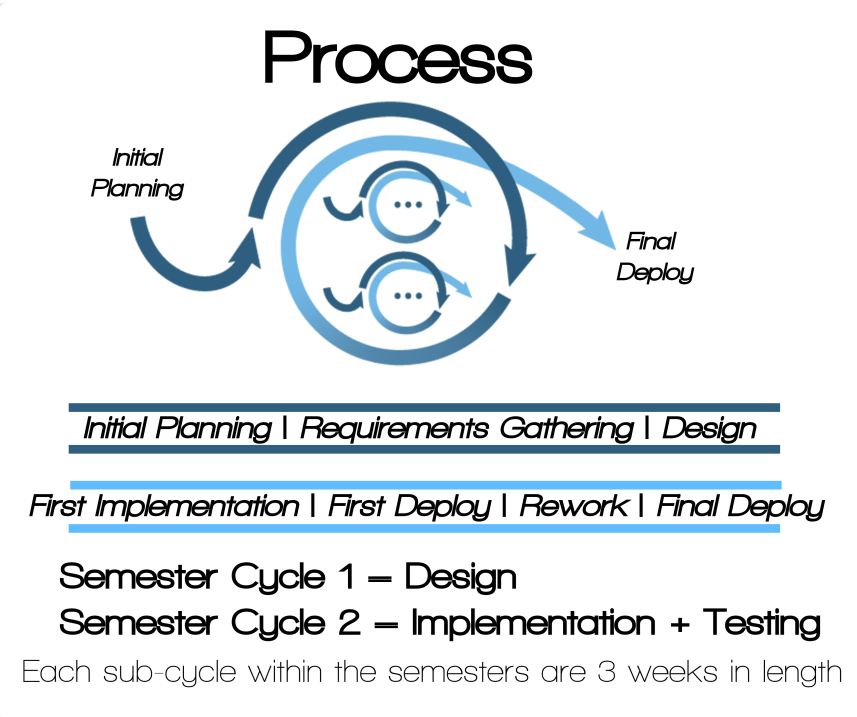
The system should be built in a way so that a new data source module can be added to the system (after being written) easily (or with less than an hour’s worth of configuration).

### Constraints

During the project lifecycle there were some constraints that had been upheld.  The biggest constraint we had was designing the product to be used in two universities with semi-unique needs.  The University of Arizona has a different infrastructure compared to RIT and therefore needed to be taken into account.  Expanding this situation to potentially include any deployment of the product we had created an architecture that allowed easy addition of functionality when and wherever it was needed.  Another major constraint was the technology stack which, for the most part, was dictated by the project sponsor.  The project utilizes the .NET framework as well as the built-in MVC framework.  The sponsor knows these technologies and wanted to be able to maintain the product with ease. Our schedule was a minor constraint in that we had other classes to attend and keep track of.  While it did not prevent the product from being completed, it did prevent use from completing all the gold plating we wanted to do. These constraints were known from the start of the project and they did not affect us very much.  In fact it was arguably easier to do the project since some decisions were already made for us.

### Development Process

The development of jobSalt followed an iterative methodology. While we did adhere to a more traditional approach with our process, we implemented certain Agile practices in our day-to-day operations in order to better handle change and risk. After developing and presenting our process to our sponsor, we had full support and we were encouraged to do whatever worked best for our team.



As the diagram above shows, our process consists of “cycles”. Each cycle has 7 steps, which are completed one after the other. Each step has an explicitly scheduled due date:

1. Initial Planning
2. Requirements Gathering
3. Design
4. First Implementation
5. First Deploy
6. Rework
7. Final Deploy

The cycles are recursive, meaning that some cycles consist of other cycles; every cycle contains the 7 steps, whether it has sub-cycles or not. On the largest scale, the project as a whole is one cycle:

1. **Initial planning** – define team roles, set up metric tracking tools, set up team collaboration tools, set up team website, define process
2. **Requirements Gathering** **–** elicitation meetings with sponsor, Vision & Scope document, SRS document
3. **Design** **–** prototype risky technologies, Architecture document
4. **First Implementation –** implement the four main features of jobSalt
5. **First Deploy –** acceptance meeting with sponsors, get website live
6. **Rework –** testing, bug fixing, finalize documentation
7. **Final Deploy –** hand off project to sponsors with appropriate documentation

The project as a whole is divided into 2 semester-cycles. The first semester is devoted to the first 3 steps of the project cycle: Initial Planning, Requirements Gathering, and Design. The second semester is devoted to the last 4 steps of the project: First Implementation, First Deploy, Rework, and Final Deploy. Each semester cycle follows the 7 steps:

1. **Initial Planning –** finalize the upcoming semester’s schedule and define deliverables
2. **Requirements Gathering –** for the first semester, this included the elicitation meetings. For the second semester, this was more focused on determining our plan of attack for implementation, and which parts of the system have the highest priority
3. **Design –** for the first semester, this included determining what documents we would need to write in order to properly document and understand the system. For the second semester, this step included assigning implementation roles according to our individual strengths
4. **First Implementation –** for the first semester, this included writing our Vision & Scope, SRS, Architecture documents, and code prototypes. For the second semester, this included implementing the 4 main features of jobSalt, along with all the framework code needed to tie it together
5. **First Deploy –** for the first semester, this included going over the documents and our understanding of the project as a whole with the sponsors, and getting feedback. For the second semester, this included demoing the project as it stood, and getting feedback.
6. **Rework –** for the first semester, this included changing our designs or understandings based on the feedback from the sponsors. For the second semester, this included tweaking functionality based on sponsor feedback, testing, and fixing the highest priority bugs
7. **Final Deploy –** for both semesters, this included handing off all the deliverables of the semester to the sponsors

Each semester-cycle is divided into additional sub-cycles, which were generally about 3 weeks in length. More of this is covered in the next chapter: Project Schedule.

**Day to Day Activities**

A big concern of our team was communication. Because of our 5th team member who worked remotely from Arizona, it was critical that we were all on the same page as a team. Also because of the very vague requirements at the beginning of the project, it became ingrained in our minds as a team to keep the project sponsor in very close communication over the course of the project.

We also followed certain quality assurance procedures to make sure that our code was always up-to-date and that our repository was clean.

**Communication**

Communication was a critical piece for making jobSalt development smooth and successful. We worked hard to maintain communication between team members, between the team and the product sponsors, between the team and the project coach, and between the RIT team members and the University of Arizona team member.

In addition to team meetings (held consistently twice a week) and emails, the team (including our University of Arizona member) communicated throughout the week using a group-based instant messaging service named GroupMe. This allowed us to discuss ideas, talk informally, answer questions, coordinate meetings, and resolve problems instantaneously, which several times noticeably removed roadblocks before they even occurred. In addition to GroupMe, the team used Google Hangouts to video chat the remote team member. The facilities in the SE team rooms made it so that it felt like the remote member was in the room.

Sponsor meetings were held consistently once a week. Except for the uncommon occasion when there was nothing important to report, we kept the sponsor up to date with all progress made on the project, resolved any ambiguity in the requirements, got feedback on our progress, and answered questions during these meetings. The fact that these meetings were so frequent is one of the biggest factors for the smoothness of the project.

We would meet our project couch at least once a week (usually during our team and sponsor meetings), but several members of the team talked with him informally throughout the week every week (this was made easy because he is the professor of several of our team members).

**Metric Tracking**

On a day-to-day basis, the team tracked the number of hours worked, what was worked on, and who worked on it. These entries were reviewed during each team meeting, which provided us with helpful feedback as the project went on. More of this is talked about in the Process and Product Metrics chapter of this paper.

**Quality Assurance**

Team Pepper used Git to manage our source code. During the Initial Planning stage of our project cycle, we decided to follow certain procedures to maintain the integrity of our Git repository. A working version of the code was always kept in the Master branch. Each team member would branch off of the master branch to make changes to the code.

At the start of every team meeting (twice a week), we integrated our branches into a special Mergebranch and immediately fixed any issues that arose as a team. When the Mergebranch was ready (meaning it built without errors and it passed some manual regression tests after running it), we would merge the Mergebranch into Master. Only one member of the team (the Project Manager) had the ability to push changes to the Master branch, which we decided after another member of the team accidentally pushed to Master and broke the build. After changes were pushed to master, the development branches were deleted, all team members took down the pushed changes to Master, and created new development branches. This was done to ensure that everyone was working off of the same code.

**Assigning Work**

Although our schedule dictated that certain work had to be done by a certain time, there is always overhead work to be done: documentation, general framework code, security code, etc. Because of this, not all team resources were necessarily spent strictly on the cycle we were currently on. Instead, work was assigned during team meetings based on what had the highest priority. For instance, during the Alumni Connect sub-cycle in the second semester, the team only really needed one developer to complete the feature. The other team members were busy finding and fixing bugs in the finished Jobs feature, or optimizing filters to work in the general framework. Finishing cycle-based work always had the highest priority, but any extra manpower was appropriately assigned (by group consensus).

### Project Schedule: Planned and Actual

Below is the actual schedule for jobSalt (keep in mind that while we knew what we had to do for the second semester during the first semester, the details of the schedule for the second semester weren’t finalized until the first week of it). Also keep in mind that only the Initial Planning, First Deploy, and Final Deploy are single-date milestones; the rest of the cycle steps are periods of activity in between those dates. A detailed explanation of the schedule is described below.

Key:

Project Cycle:

08/23/2013 – 05/212014 (school year)

Semester Cycle:

08/23/2013 – 12/12/2013 (First Semester), 1/28/2014 – 05/21/2014 (Second Semester)

Sub Cycle:

Roughly 3 week increments (exact dates shown below)

1. **Initial Planning – jobSalt and First Semester Aug. 23 – Sept. 3**
2. **Requirements Gathering – jobSalt**
   1. **Requirements Gathering – First Semester**
   2. **Design – First Semester**
   3. **First Implementation – First Semester**
      1. **Initial Planning – Sub-Cycle 1 (Vision & Scope) Sept. 3**
      2. **Requirements Gathering – Sub-Cycle 1**
      3. **Design – Sub-Cycle 1**
      4. **First Implementation – Sub-Cycle 1**
      5. **First Deployment – Sub-Cycle 1 Sept. 12**
      6. **Rework – Sub-Cycle 1**
      7. **Final Deployment – Sub-Cycle 1 Sept. 19**
      8. **Initial Planning – Sub-Cycle 2 (SRS) Sept. 24**
      9. **Requirements Gathering – Sub-Cycle 2**
      10. **Design – Sub-Cycle 2**
      11. **First Implementation – Sub-Cycle 2**
      12. **First Deployment – Sub-Cycle 2 Oct. 3**
      13. **Rework – Sub-Cycle 2**
      14. **Final Deployment – Sub-Cycle 2 Oct. 17**
3. **Design – jobSalt**
   * 1. **Initial Planning – Sub-Cycle 3 (Architecture) Oct. 22**
     2. **Requirements Gathering – Sub-Cycle 3**
     3. **Design – Sub-Cycle 3**
     4. **First Implementation – Sub-Cycle 3**
     5. **First Deployment – Sub-Cycle 3 Oct. 31**
     6. **Rework – Sub-Cycle 3**
     7. **Final Deployment – Sub-Cycle 3 Nov. 28(1)**
     8. **Initial Planning – Sub-Cycle 3 (Prototypes) Nov. 05**
     9. **Requirements Gathering – Sub-Cycle 3**
     10. **Design – Sub-Cycle 3**
     11. **First Implementation – Sub-Cycle 3**
     12. **First Deployment – Sub-Cycle 3 Nov. 14**
     13. **Rework – Sub-Cycle 3**
     14. **Final Deployment – Sub-Cycle 3 Nov. 21**

1. **First Deployment – First Semester Nov. 28**
2. **Rework - First Semester**
3. **Final Deployment - First Semester Dec. 12**

**~~~ Winter Break ~~~**

1. **First Implementation – jobSalt**
   1. **Initial Planning – Second Semester Jan. 28**
   2. **Requirements Gathering – Second Semester**
   3. **Design – Second Semester**
   4. **First Implementation – Second Semester**
      1. **Initial Planning – Sub-Cycle 1 (Job Search) Jan. 28**
      2. **Requirements Gathering – Sub-Cycle 1**
      3. **Design – Sub-Cycle 1**
      4. **First Implementation – Sub-Cycle 1**
      5. **First Deployment – Sub-Cycle 1 Feb. 06**
      6. **Rework – Sub-Cycle 1**
      7. **Final Deployment – Sub-Cycle 1 Feb. 20(2)**
      8. **Initial Planning – Sub-Cycle 2 (Alumni Connect) Feb. 25**
      9. **Requirements Gathering – Sub-Cycle 2**
      10. **Design – Sub-Cycle 2**
      11. **First Implementation – Sub-Cycle 2**
      12. **First Deployment – Sub-Cycle 2 Feb. 27**
      13. **Rework – Sub-Cycle 2**
      14. **Final Deployment – Sub-Cycle 2 Mar. 06**
      15. **Initial Planning – Sub-Cycle 3 (Salary) Mar. 11**
      16. **Requirements Gathering – Sub-Cycle 3**
      17. **Design – Sub-Cycle 3**
      18. **First Implementation – Sub-Cycle 3**
      19. **First Deployment – Sub-Cycle 3 Mar. 20**

**~~~ Spring Break ~~~**

* + 1. **Rework – Sub-Cycle 3**
    2. **Final Deployment – Sub-Cycle 3 Apr. 03**
    3. **Initial Planning – Sub-Cycle 4 (Housing) Apr. 08**
    4. **Requirements Gathering – Sub-Cycle 4**
    5. **Design – Sub-Cycle 4**
    6. **First Implementation – Sub-Cycle 4**
    7. **First Deployment – Sub-Cycle 4 Apr. 17**
    8. **Rework – Sub-Cycle 4**
    9. **Final Deployment – Sub-Cycle 4 Apr. 24**

1. **First Deployment - jobSalt**
   1. **First Deployment – Second Semester**
2. **Rework - jobSalt**
   1. **Rework – Second Semester**
      1. **Initial Planning – Sub-Cycle 5 (Testing) Apr. 29**
      2. **Requirements Gathering – Sub-Cycle 5**
      3. **Design – Sub-Cycle 5**
      4. **First Implementation – Sub-Cycle 5**
      5. **First Deployment – Sub-Cycle 5 May 08**
      6. **Rework – Sub-Cycle 5**
      7. **Final Deployment – Sub-Cycle 5 May 15**
   2. **Final Deployment – Second Semester**
3. **Final Deployment – jobSalt May 21**

The planned schedule of jobSalt versus the actual schedule is very similar, with a few notable points and changes. The first thing to note is that at the very beginning of the first semester (08/23 – 09/03) we did not yet have a process or a schedule yet, so we determined that to be the initial planning phase of both the project as a whole and the first semester. This is also why the first semester has only 4 sub-cycles while the second semester has 5.

The second thing to note is that the Initial Planning phase of a sub-cycle does not start the day after the Final Deployment phase of the previous sub-cycle. This is because our cycles start and end always on the day of a team meeting: Initial Planning is always on a Tuesday (first meeting of each week) and Final Deployment is always on a Thursday (last meeting of each week, and also the day of our sponsor meetings). During the gaps in between defined cycle activities, the team usually worked independently on bugs or other development odds-and-ends from previous cycles, or took a break if nothing super important needed to be done.

The third thing to note is how we ordered our sub-cycles. As is written in the Basic Requirements chapter of this document, different features had different priority levels. We knew that we could not determine with 100% accuracy how much time any of them would take, so we made sure that the more important features were done first. The less important features were stacked towards the end, and as a team we were willing to (in the worst case scenario) take time away from the less important features to make sure the high priority ones were done well. The priorities of the features were determined during meetings with our sponsors, and they were fully on board with our scheduling contingency plan. This plan wound up being very helpful when we ran into a scheduling slip with the Jobs feature (explained below) and Spring Break, which we accidentally didn’t initially plan for.

In terms of schedule changes, only two major ones took place. They were never impulsive or sudden; if we ever modified the schedule of a cycle, we made sure the sponsors knew before the cycle started that the schedule may need to change based on risks that the team identified.

The first instance of this happening (noted above with a (1)) was during our Architecture sub-cycle. We knew going into the sub-cycle (based on information gleaned after writing our SRS) that our architecture would be very hard to define before we had more experience with the technologies we planned to use. We told our sponsors this going into the cycle, and explained that if we hit any road blocks, we may start prototyping early and wait to finish the architecture until after we were finished. After the first draft of the architecture was released, we realized we could not feel comfortable with any architecture we came up with until we got our hands on certain technologies to see how they interacted. Instead of starting our Rework phase right then, we launched right into the Prototyping cycle with our sponsors approval. After that cycle was complete, we felt comfortable jumping back into the Rework and Final Deployment of our architecture, and there were no schedule slips that came because of the switch.

The second instance of a schedule change happened at the end of our Jobs implementation. We were not sure of the amount of framework development that would need to happen at the beginning of the project. The team realized that we underestimated the amount by the third week of the cycle, and decided to add another week to pump more quality into the flagship first feature. The sponsor was fully aware that this might happen, and was on board with our decision. We took the week from the Alumni feature, which wound up being so similar to our Jobs feature that we more than made up for the time lost.

### System Design

One of the basic requirements of jobSalt was that it had to be a website. Because of that, and because of the complexity of the data the system had to handle, we approached jobSalt as an enterprise web system. As such, the system has three layers: Presentation, Domain, and Data Source.

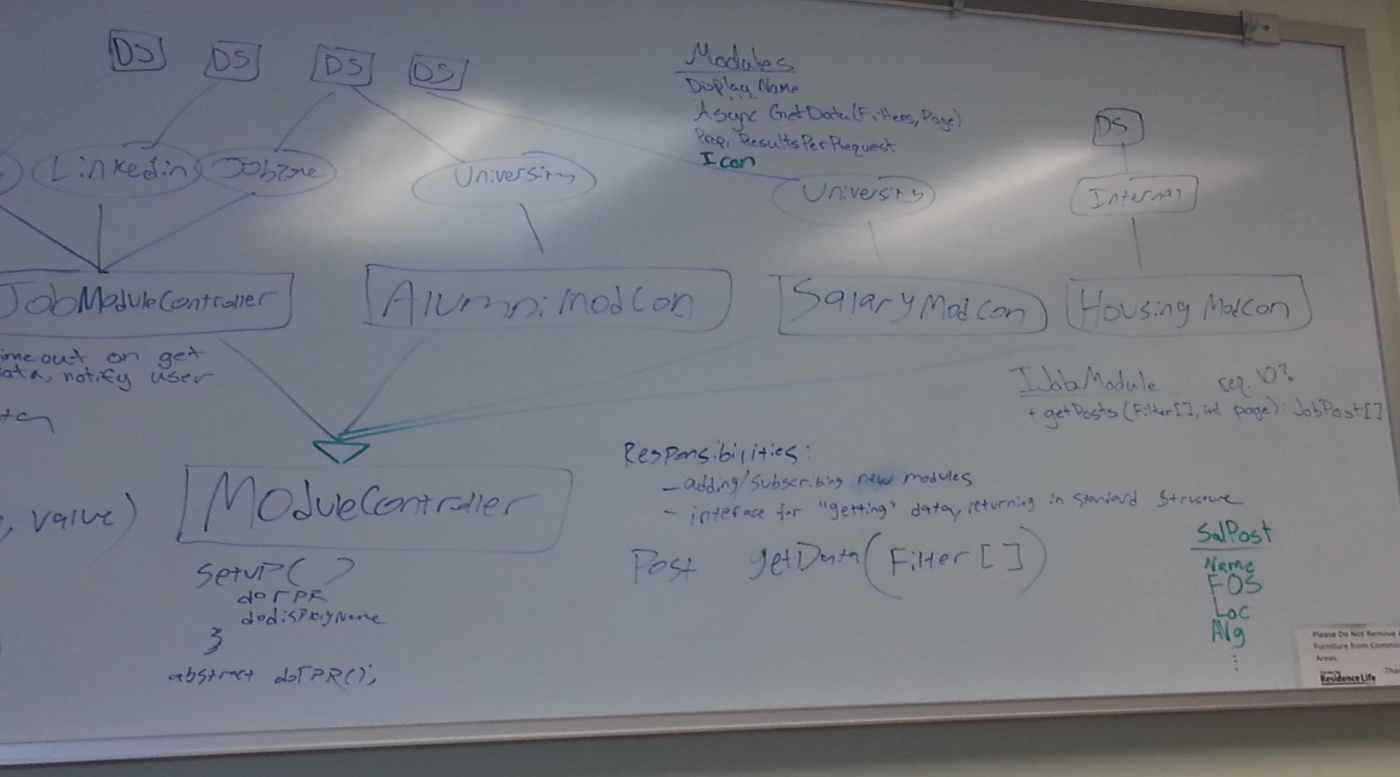
Our Presentation layer consisted of the user interface and the various inputs the user could make into the system.

Our Domain layer consisted of our Shepherds and how we handled and manipulated the data coming in.

Our Data Source layer is arguably the most complex and detailed layer of our system, and where the vast majority of our work was put. This layer is responsible for collecting the data from external sources, which is the most important part of our system; without the data, jobSalt does very little.

**Data Source / Domain Architecture**

The jobSalt system is built with a modular architecture. This architecture was decided on after several iterations. Here is a whiteboard from the start of our design phase, with an initial idea of how to design our system:

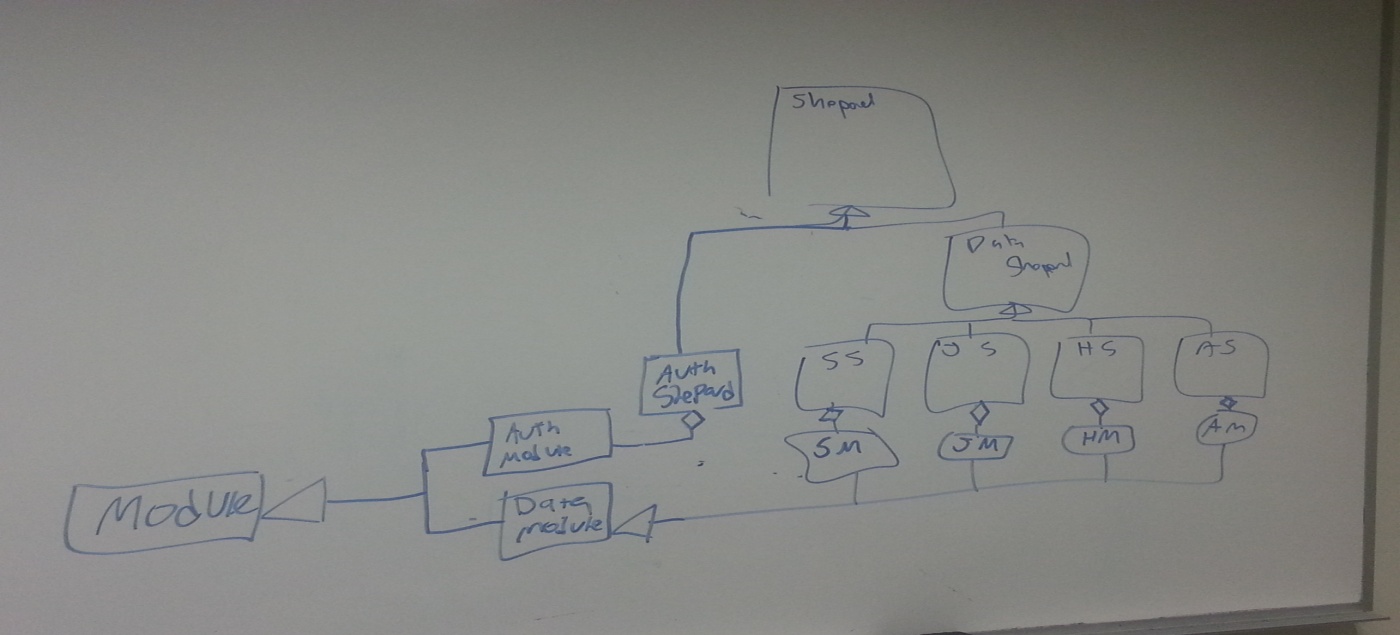


As the picture shows, we knew from the start that a modular architecture was the best approach. We reached this conclusion based on a few factors:

* There were several data sources that our sponsors wanted to pull job posting information from. These sources had different formats, but they needed to be displayed in a consistent way in our system
* We knew that in the future, our sponsors would want to add more data sources without too much trouble
* For each main feature, we as a team could conceive of multiple sources being fed into the system at some point in the future, even if we only had one source for the first release. For instance, even though we were only pulling Alumni data from the University’s database for the first release, we could conceive of LinkedIn supplementing that information in the future and we wanted to design a system that could easily handle that

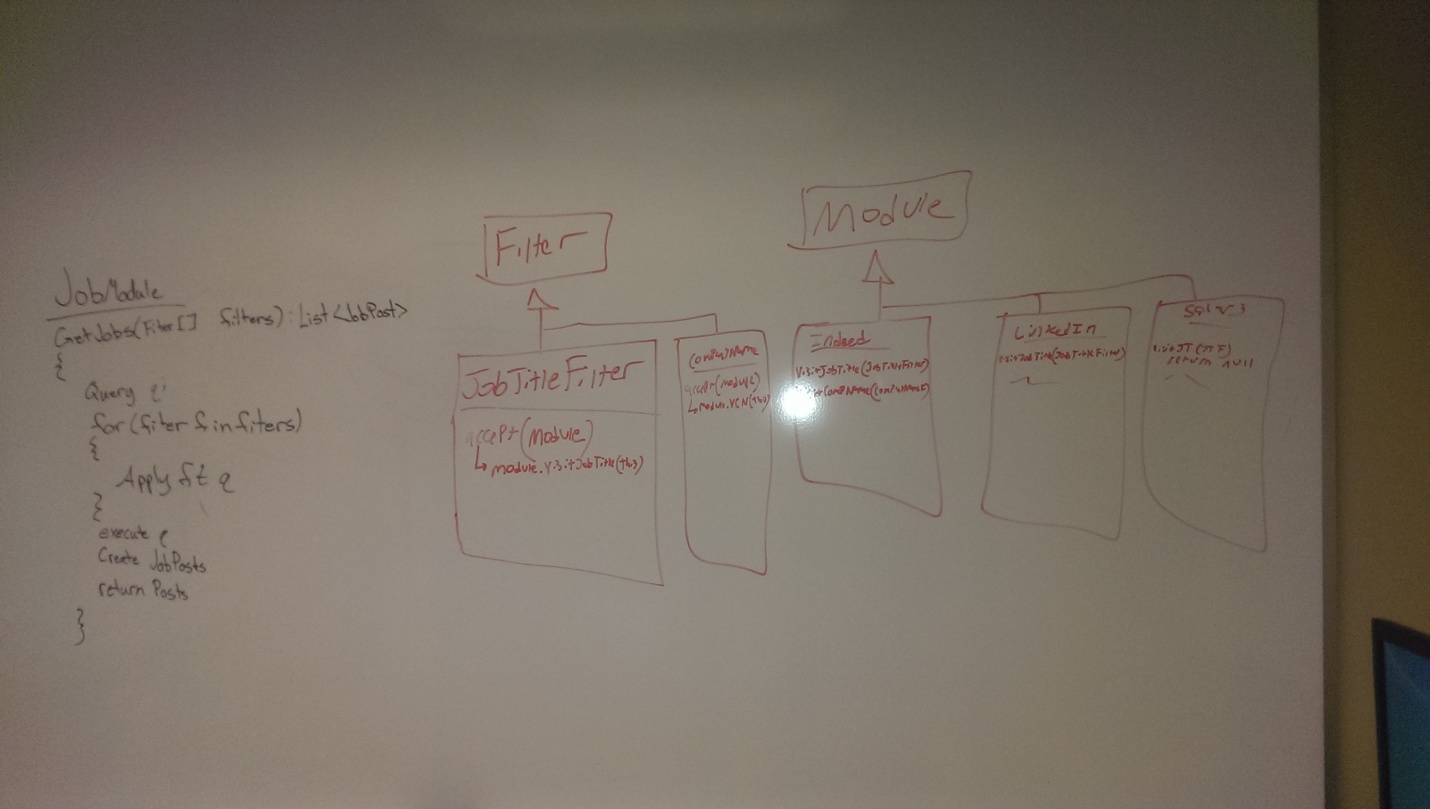
At the beginning of our design, we had “modules” pulling from “data sources” (labeled DS in the picture). These modules fed into a “ModCon” (or Module Controller), each of which inherited from a ModuleController interface. A Module Controller was responsible for subscribing to any new modules to retrieve results. It was also responsible for “giving” the modules an array of “Filters” so that the modules would only retrieve relevant data. Each module inherited from one Module interface, which we would expose to third party developers so they could write their own modules. A module is responsible for having a display name, an icon, and (most importantly) a method for retrieving data from the source it is tied to.

Over time, we realized that our authentication needs matched our data needs fairly well. We wanted to support multiple types of authentication (CAS, LDAP, etc.), so we thought “why not have each type of authentication be contained in its own module, handled by a module controller?” By this point we had also come to think of the modules as “sheep” that were being “herded” by the Module Controllers… to limit the vocabulary confusion between Module Controllers and Page Controllers (the latter is used by MVC to execute system actions based on a user’s URL), we started called the Module Controllers “Shepards” (a misspelling of “shepherd” which we didn’t actually realize until very late in the project). Because of this, we came up with the idea of an Authentication Shepard, and separated it from our data shepards.



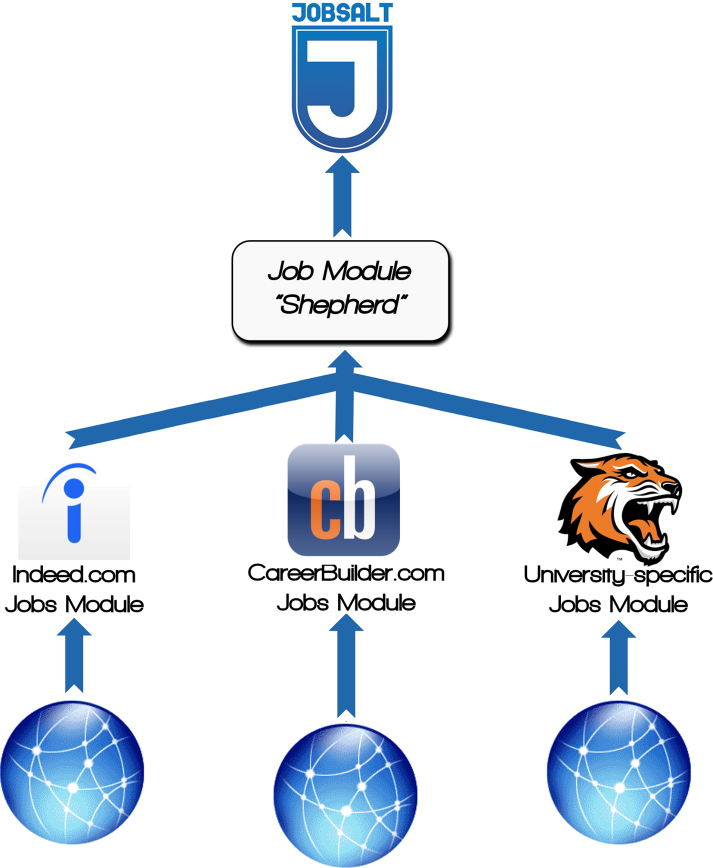
At this point, all of our modules still inherited from a general Module interface. The data section was slightly different than the auth section of our system, and so they were not mixed together, but they worked in basically the same way. Importantly, in this version of our system we still had a “super shepard” which all the shepards inherited from, and then a “data shepard” which inherited the super shepard and was the superclass of only the data shepards. In later versions of our architecture we realized this was unnecessarily complicated, and took out the concept of a “data shepard”.

Below, we can see a later revision of our architecture that touches on a few key concepts:



First is how we reorganized our superclasses. We got rid of the idea of one overarching “module” interface when we realized that the different features had different requirements of their modules (most critically, Job Modules need to return a list of JobPosts, Alumni Modules need to return a list of AlumniPosts, etc.). Because of this, each feature got their own module interface that would be exposed to third party developers. In this picture we can also see an early design of our Filter system, which would wind up getting refactored several times.

In the end, we decided on the following architecture that would be consistent across all features (below is an example of the Job Postings feature):



Notably, there is one Shepard per feature, which is tied to one MVC controller. The Shepard is the gateway into the data source, so that anything above the Shepard has no knowledge or control over modules or data sources. Shepards no longer inherit from a superclass, because there’s no time we ever want the Shepards to be lumped together; they are completely separate from each other and represent totally discreet sections of our data source layer. Also, all of the modules for a given feature inherit from an interface specific to that feature. In other words, all the Job modules inherit from IJobModule, all the Alumni modules inherit from IAlumniModule, and so on. Authentication continues to be its own “feature”, and acts in almost exactly the same way as the other features. There is an authentication shepard, which attempts to authenticate with each of its sheep modules until one succeeds or they all fail.

### Process and Product Metrics

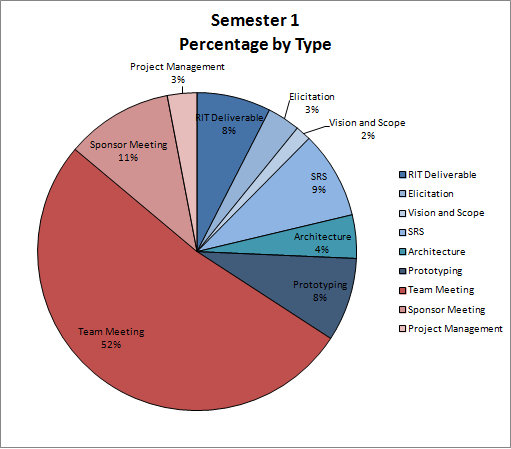
The raw data for the time-based metrics were collected through a Google form that, when submitted, automatically updated an online spreadsheet. The information for each time log entry included Name of Team Member, Date of Work Done, Type of Work Done (selected from a list of pre-defined values that the team’s project manager decided), and Length of Time Working (in hours and minutes). Each team member, after working on a certain portion of the project, would manually submit his hours through this form.

**Time-Based Metrics**

It was possible from the raw time-based measurements to calculate interesting metrics about the type of work that was done on a day-to-day, week-to-week, and semester-to-semester basis.

Total Work Breakdown by Type

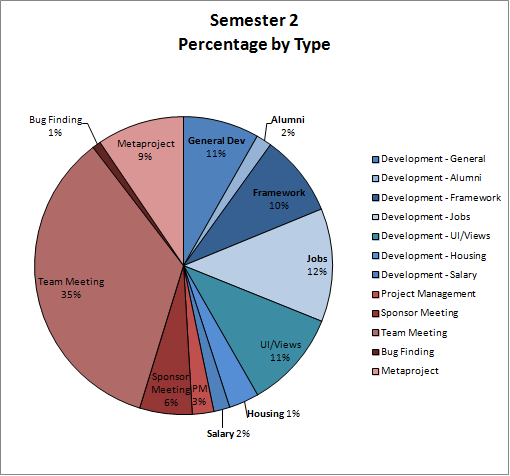
The first set of graphs to show is how we distributed our time to the various pieces of the project. Red shades of the graph represent meta-work such as meetings and project management. Blue shades of the graph represent concrete deliverables like project code and documents:



Semester 1

Cumulative Work Time: **240:25** (h:m)

An important thing to note when interpreting this graph is our decision to log any work done on deliverable during meetings as “Team Meeting” instead of logging it for the deliverable we were working on. Because of this, our meeting time seems very high; much of the time during those meetings, however, was spent interpreting our elicitation results as a group and generating requirements documents. We felt this had to be done in person as a group to achieve the highest level of understanding as a team. The “Project Management” section combines several activities, including formatting time logs, setting up JIRA, setting up group communication, and so on. The “RIT Deliverables” section combines deliverables and activities that were required by the SE department, such as the project synopsis and project presentations.



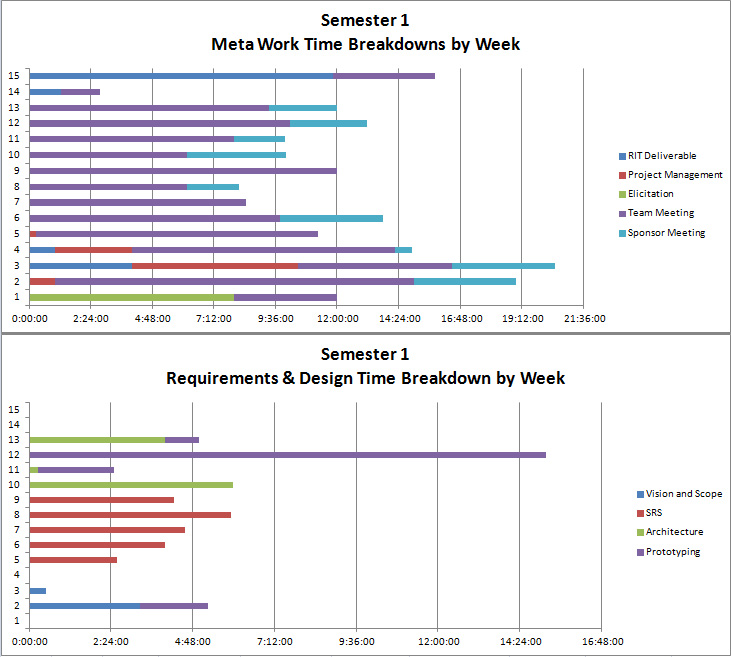
Semester 1

Cumulative Work Time: **400:55** (h:m)

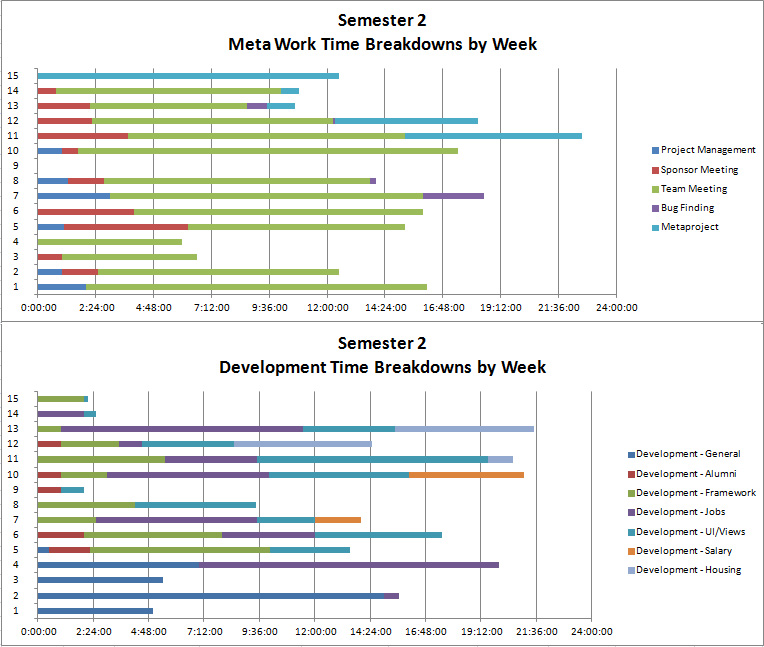
There are several important differences between the graph for the first semester and the graph for the second semester. Importantly, the gap between “Meta Work” (shaded in red) and “Implementation Work” (shaded in blue) is much smaller in the second semester than the first. This is partly due to the fact that more work could be done independently outside of meetings. The “Metaproject” section (a specific subsection of Meta Work) combines several activities and deliverables similar to the “RIT Deliverables” section in the first semester. These include making the project poster, preparing for the project presentation, and doing various documentation work (including writing this report). Another important thing to note is the differences in cumulative work time which in the second semester is almost double that of the first semester. It’s also interesting to see that while the Jobs section of our implementation took about 12% of the total work time of the semester, the other features combined (Housing, Salary, and Alumni) take up only 5%. This is because our features work very similarly to each other; after coding the one feature, it was almost negligible to write the others. Finally, it’s important to pay attention to the section titled “General Development.” This is reflective of the difficulties we encountered logging our times for the beginning of the semester. Not only were the logs inaccurate, but we had not specified how granular we wanted the log descriptions to be. Because of that, all the work we did was lumped into one general category. This is unfortunate, but we recognized that our logging procedures at the time weren’t working and we took steps to correct it. In this sense we were successful.

Total Work Breakdowns by Week

Having data for the work done over the entire semesters is interesting, but what about the work we did per week? Our schedule was made for a week-to-week basis after all, so it’s useful to see how much our work was on schedule for those weeks. It’s also useful to see how much “overhead” was done alongside the actual development work:



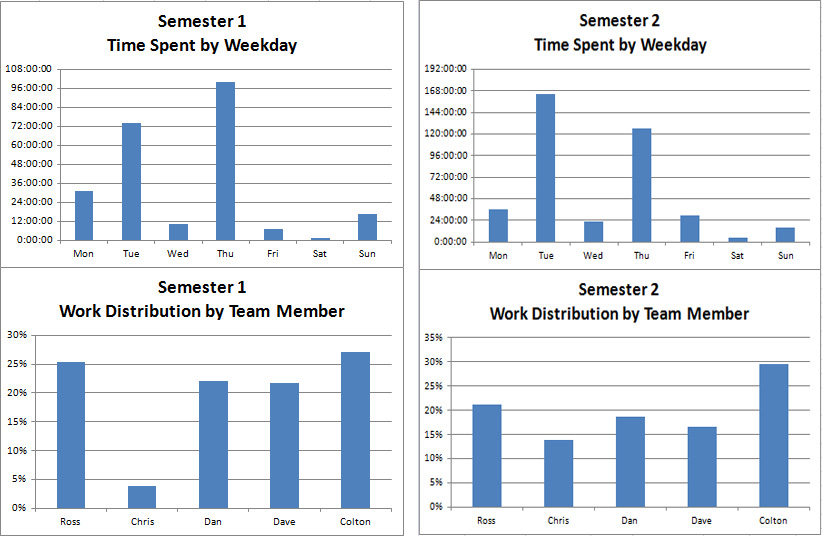
Before interpreting these results, make sure to note the scale differences between the two charts; they are not the same. It’s interesting to note a few things on the above graphs. First, most project management tasks were done early on in order to automate most of the PM work. We consistently had sponsor meetings every week, and the majority of our time mid-semester was on the (formidable) SRS document. It’s clear how we worked on our Architecture and then paused to prototype, and then finished up our Architecture (as is explained in an above chapter). Most of the work done at the end of the semester was focused on our presentations and RIT deliverables.



The graphs for the second semester are quite different from the first. Again it is clear that we met with the sponsor on a very regular basis, as well as held team meetings. Project management was done consistently throughout the semester, in an attempt to modify our time logging process and develop an accurate bug log. Framework and UI/Views development was a constant in terms of our implementation semester, as well as improving our Jobs page (our highest priority feature). Just like the first semester, most of the meta-project work (such as presentations and documentation) was done towards the end.

Other Interesting Data

These graphs show our work breakdowns by weekday, as well as by team member:



These graphs are pretty self-explanatory. Tuesdays and Thursdays were our busiest days of the week (which makes sense because we met as a team and with sponsors on Tuesdays and Thursdays).

The work breakdowns by team member are also very interesting. In the first semester the RIT team worked a very similar amount, but we had trouble working with our remote team member in Arizona. In the second semester, once we ironed out communication issues and built trust as a team, we did a much more even distribution of work. Each of the 5 team members are at about 20% by the end of the project, which we consider to be a success.

### Product State at Time of Delivery

The jobSalt system at the time of delivery is a complete product and has implemented the four main feature branches: job search, alumni connections, salary information, and housing reviews. Aspects of jobSalt that are not yet fully implemented at the time of release are listed below.

**Twitter**

The original intent was to use twitter to gather job postings as many companies are posting openings on these type of social media outlets. After inspecting the API of twitter as well, as

third party API’s that specifically try to get jobs from Twitter, we determined that it was not accurate enough. There was no good way to ensure a tweet was indeed a job post and no good way to filter by location, company, job type, or job title. If we were to include it into the jobs feature then the perceived accuracy of the system would be lessened. As a compromise with the sponsors it was chosen to include the OCE Twitter feed into the website. This provides users with useful university related tweets without compromising the accuracy of the job search feature.

**LinkedIn**

LinkedIn from the beginning was intended to be one of our main sources for job postings as well as the main source for alumni. The endpoints within the system were implemented to convert job postings data from LinkedIn’s types to our own internal type. Simultaneously to developing the LinkedIn feature we also submitted a request to gain access to their API, and more specifically there new API program for colleges that allows access to alumni. We did not hear back from LinkedIn for several weeks and at that point received a negative response. Our sponsor at this point reached out to some contacts he had within LinkedIn but unfortunately this did not get us any further with LinkedIn. Without access to the LinkedIn API we were unfortunately unable to include it into the job postings feature or the alumni feature.

**Shibboleth**

Our main authentication strategy was to use Shibboleth. We had a lot of difficulty getting Shibboleth to work in the system though. We envisioned the login system for jobSalt to be contained within the system and not redirect to another site for authentication. This make implementing Shibboleth much harder, despite that though we were able to make good progress in doing just that against a test Shibboleth end point. At the same time our sponsor had also requested, from ITS, the needed information to connect to RIT’s implementation of Shibboleth. It wasn’t until the last week in the second semester that we got a response and it was still not helpful at that point. Unfortunately due to the high number of hours we were putting into just the Shibboleth feature combined with an already functioning LDAP authentication feature we decided to cut our loses and freeze development of Shibboleth and focus on other aspects of jobSalt.

**University of Arizona Modules**

We faced some difficulties when gathering the data from University of Arizona (UA). Both RIT and UA use Symplicity in their OCE departments for providing he job posts to students, but RIT backs up the data to a SQL database where as UA does not. For RIT we interface with the SQL database as it gives us much more control. For UA Chris Ray wrote a web service that interfaces with Symplicity (which only has a reporting API) and provides a restful end point to get job postings. This solution works for getting UA job postings, but UA does not store their salary or alumni information within Symplicity which prevented us from getting that data in the same manor. Furthermore UA does not keep their Alumni or Salary data on campus which prevents us from gaining access to that information at all. Until this situation changes we are unable to provide UA with the Salary or Alumni modules for their university.

The following features were not given as requirements but are additional features we were able to implement for the time of release.

**Dice, Github, and CareerBuilder Modules**

Chris Ray was able to find and implement several job modules that we did not find ourselves. He did this in part to test out how easy/difficult it is to add a new source to jobSalt. Each module only took him a few hours to implement which is a good sign for expandability of jobSalt. The CareerBuilder module has also became one of our main three job sources.

**Duplicate Job Post Removal**

We implemented a system that matches duplicate posts from different sources and removes the duplicated post. For instance if a company posts the same job to both Indeed and CareerBuilder our system can match that the post has already been showed to the user and remove it before it is presented to the user.

**Configuration**

Originally we were intending to just provide configuration files that could be modified by the admin on the machine. In the end we were able to move that functionality into a page of jobSalt that only shows up for admin users.

### Project Reflection

This project overall went very well. The primary reason our project was so successful is that we had a great team. Everybody was pulling their weight, and we had great communication. Throughout the project we were meeting twice a week very regularly. In the first semester we spent a lot of time together just working, as a lot of the documents and work done in the first semester had to be done as a group. We were very good about doing this in our two weekly meetings. To augment our communication we also used a mobile based communication medium called GroupMe. We had very regular communication on this practically 24/7. Both these meetings and this application were also used with our group mater in Arizona. Though we had some hiccups communicating with him during the first semester, these were ironed out, and both sides of the team started working very well together.

Though our schedule worked out very well, we had a few things we would have liked to have changed if we were to redo this project. For this project we really likes how we separated the two semesters, The first semester spent on planning and design was very useful. That being said, we felt the time meant to be spent on architecture was not very productive, and it would have been more efficient to have spent that time doing more prototyping.