weather.rit.edu

The Weather Hub
Conor Craig, Chris Norman, Zack Martin, Matthew Stevens

Project Sponsors
William Springer, James Stefano

Faculty Coach
Larry Kiser
Project Overview

The weather.rit.edu project aims to provide a website that allows members of the RIT community near real-time access to the weather conditions directly on campus. In addition to providing users a portal with which to check the conditions at RIT, the weather data that the project collects will be archived and allowed to be accessed and searched. Third-party weather services such as alert feeds, forecast, and radar information will also be provided by the website.

Basic Requirements

The main functional requirement of the system was to display accurate and up-to-date weather information on RIT’s campus to the user. The weather information is provided to the server by sensors located on RIT’s campus. This information appears to the user as one source of data, however the system is designed to read data from multiple sources. The primary user profile for the system is a member of the RIT community who is interested in the current weather information on campus. The primary use case, viewing current weather data, should take as little input as possible - accessing the website is the only step the user needs to take to view current weather data.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Input</th>
<th>Output</th>
<th>Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system shall collect weather data from external software interfaces.</td>
<td>NWS API</td>
<td>Weather information displayed on the website</td>
<td>None</td>
</tr>
<tr>
<td>The system shall collect weather data directly from RIT sensors.</td>
<td>WeatherLinkIP</td>
<td>Weather Data cached every 2 minutes</td>
<td>None</td>
</tr>
<tr>
<td>The system shall archive the preprocessed data.</td>
<td>Weather data from the scheduler module</td>
<td>Weather data saved to the database</td>
<td>None</td>
</tr>
<tr>
<td>The system shall display information via a web interface.</td>
<td>Weather data from the cache</td>
<td>Weather data represented on the website</td>
<td>User accesses the website</td>
</tr>
<tr>
<td>Feature</td>
<td>Action</td>
<td>Outcome</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------------------------------------</td>
<td>--------------------------------------</td>
<td></td>
</tr>
<tr>
<td>The system shall allow for future sensors addition.</td>
<td>New sensor added to the system</td>
<td>New source of weather data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developer writes adapter for sensor</td>
<td></td>
</tr>
<tr>
<td>The system shall allow for searches of archived data.</td>
<td>Date ranges</td>
<td>System displays results of search</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>User provides date ranges through website</td>
<td></td>
</tr>
<tr>
<td>The system shall export data in the form of comma separated value text.</td>
<td>Date ranges</td>
<td>System sends a CSV file of all data for the given date range to the user’s web browser</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>User provides date ranges through website</td>
<td></td>
</tr>
<tr>
<td>The system shall integrate an image feed from a webcam, either FTP file or live video stream.</td>
<td>RIT Frisna Quad Webcam</td>
<td>Webcam feed displayed on website</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

**Constraints**

Once the team was able to solidify the hosting environment, we were able to choose the technologies we want. The only constraint imposed the hosting environment was we had to use Redhat linux. That meant any technology that we would use would have to be compatible with that flavor of linux. Since Redhat is pretty versatile, most open source web technologies favor a linux based environment. So, picking a programming language to develop in would be an easy task.

Another constraint was we had to use the WeatherLinkIP hardware in order to receive weather information from the Davis sensor. Dealing with that constraint proved to be troublesome at times. There were periods when the sensor would stop responding or go down, which meant we were unable to receive weather data for certain periods of time.

**Development Process**

Evolutionary delivery was used for the entirety of the software lifecycle.
The sponsors were very clear in their stance that all development issues were to be handled internally by the developers; the sponsors had no input on the process method selection. A sponsor meeting occurs once per week. This weekly meeting was appropriate for the software conceptualization, requirements elicitation, and design phases of the evolutionary delivery. Once the development team engaged in implementation sprints, the weekly meetings were cut back to meeting every two or three weeks.

Roles for the team were identified in the first weekend meeting of the first semester. The final (at the end of the project) roles for the team broke down into a project coordinator, a system administrator, and two generic developers. The roles began with one project coordinator and three generic developers; however, due to the web-centric focus for this project one team member had to take the lead on system administration.

**Project Schedule: Planned and Actual**

The scheduling for the project was split between the two semesters. The first semester was planned out ahead of time for software conceptualization, requirements, and design; the second semester was adaptive to the needs of the sponsors for the implementation sprints. The key activities for the first semester were:

1. Software conceptualization
2. Requirements elicitation
3. R1 delivery
4. R2 delivery

The key activities for the second semester were:

   1. R3 delivery
   2. R4 delivery
   3. Sensor freeze
   4. Feature freeze

The schedule for the first semester was completely broken by week 5; project-crippling hardware and hosting environment issues prevented the team from moving past the requirements elicitation phase. Requirements phase was solidified and done by week 5 and then the next 6 weeks were spent resolving if the project had hardware weather sensors and a hosting environment for the website. The team spent the down-time engaged with user-interface requirements. A R1 delivery was done in the first semester that encompassed a proof-of-concept product that connected to the weather sensor and pulled data from it. An R2 delivery occurred over the winter-break between semesters which contained necessary software architecture components such as the database layer.

Due to scope breakage suffered in the first semester, the schedule for the second semester was adaptive to the weekly sponsor correspondence meeting. The evolutionary delivery became commitment-based where all stakeholders understood that the foundation for the system had to be moved in before anything else could be developed. A part of this project is integrating hardware weather sensors to the software, and there was a cutoff for the sponsors delivering new sensors to the team which was in week 8. Feature freeze occurred in week 12, though the team was not code-frozen until week 15.

System Design

The weather.rit.edu system uses a basic MVC approach. The weather data is represented as our models. The views are represented by HTML templates that take the model information and display it to the end user. The data layer serves as a means to retrieve and store data into the databases. MVC was chosen because it was an architectural pattern everyone in the group was familiar with and it made sense for the style of web application being built.
Figure 1: System Architecture

Presentation Layer

This layer is used to present the client with weather data. This layer uses a templating engine called “Linkedin Dust.js”. Linkedin Dust.js is a HTML templating engine that can be used on both the server and client side. The benefit to having the ability to do both, is the application can render the overall page from the server and any smaller pieces on the client. Dust.js also allows templates to be compiled down to javascript for faster client side rendering. That allows the application to be extremely flexible in how it handles html rendering.

There are also other templating engines such as handlebars.js, backbone.js and Angular.js. Each of those templating engines are powerful and have the ability to do what the application needs, but in the end the team chose Linkedin Dust. In order to decide which ones were applicable to the application a process of elimination was used. The team closely examined each of them and assessed the pros and cons of each.

We didn’t choose Angular.js because the application doesn’t have much front end functionally, rather most of the heavy lifting work is done in the backend. Angular.js is more suited for applications that require a large amount of functionality on the frontend. Angular.js can only be used on the front end, so you lose the ability to perform any server side rendering.

Handlebars.js and Backbone.js are similar to Linkedin Dust.js. The downside to them is they can only be used on the client, whereas Linkedin Dust.js can be used on both the client and the server in order to render pages.
Using LinkedIn Dust.js allows the application to take advantage of two view rendering design patterns “Two Step View Rendering Pattern” and the “Composite View Rendering Pattern”.

In order to achieve both flexibility and speed the team decided to implement the Two Step View Rendering pattern. The first step is supplying the Dust.js template with some form of data. That data is provided in a JSON format. Once the request for the data is made, it is then populated within the template and logical and control operators such as if statements and loop iteration are applied to the formatted data.

After the data has been populated in the Dust.js template the second stage then begins. In the second stage the Dust.js template is rendered into HTML and CSS is applied. The template is then presented to the user as a fully styled HTML page.

**Figure 2: Two Step View Rendering**

Another benefit that can be achieved from using HTML templates is the ability to take advantage of the composite pattern and apply that to rendering multiple views at different times of the web pages life cycle. Dust.js has the ability to nest templates within other templates. That allows you to break view templates into small pieces allowing for better maintainability.

**Business Layer**

The business layer contains all the application logic. This part of the system is composed of all the server side logic for manipulating the weather data. This layer is also responsible for managing the weather sensors and gathering data from outside APIs.
This part of the system is built entirely with Node.js. Node.js is an asynchronous javascript framework. One benefit to using Node.js is that it is JavaScript. Which means the learning curve is as steep if you are already familiar javascript. Also Node.js uses non-blocking IO, which means it can handle many concurrent connections. Node.js also have the ability to be clustered. That means you can run a node process based on the number of CPUs on the server. By using that feature the application can increase the number of concurrent users without having to increase the power of your server. Also Node.js has a large open source community. Its simple to find modules to add functionality a web application. The downside is a module might not work as intended. In some cases there can be trial and error when using open source modules. Node.js also plays nicely with Mongoddb, which is the database being used in the application.

When examining Node.js research was also done on other web frameworks. There are many languages such as ruby, python, and php that are used to build web applications. As a team, we discussed the advantages and disadvantages each and came to a few conclusions. Php or python wasn’t chosen because most of the team was not familiar with building web applications in those languages. The ramp up time required to develop in those languages would take a substantial amount of time. Both php and python web applications can be difficult to maintain.

When building applications with ruby the default web framework is Ruby on Rails. At first the team considered Ruby on Rails because we all have had some exposure to it. Since we all had exposure to it the ramp up time would not take as long as python or php. The community for Ruby on Rails is also large and the framework has been proven in many production environments. Ruby on Rails was fazed out in favor of Node.js. Setting up a real-time system in Ruby on Rails seemed cumbersome, whereas Node.js seemed to fit that bill more accurately. Deployment of a Node.js application is simpler as compared to Ruby on Rails.

The web framework we ended up choosing for Node.js was Express.js. Express.js is a lightweight web framework for building web applications. The reason we went with that web framework is because it is mature and has been thoroughly tested. The documentation is comprehensive and community is large, so resolving issues is a simpler task.

There is also a framework called “The MEAN” stack. It stands for Mongo, Express, Angular, and Node.js. It is similar to a LAMP stack, but the MEAN stack is designed to work with Node.js.

Instead of using a pre existing web stack, such as the “MEAN” stack, our own custom web stack was constructed. The team did that because we wanted to have more control over what technologies would be in our web stack. Another reason we decided to build up our own web stack is we wanted to have the ability to switch out different technologies as we saw fit. If we were to use the MEAN stack it might of been harder to switch out certain components because we don’t know how deeply each technology is integrated within its system.
Caching

An issue the application had early on was the hardware used to retrieve weather information was unreliable. There were times when data took a long time to be received, to the point where the site would not load. In order to fix that problem a caching system was implemented. The goal of the caching system would be to store the latest weather data and have the site pull from the cache instead of sending a request to the hardware each time the site loads.

**Figure 3: Caching System**

![Caching System Diagram](image)

When designing the caching system, there was a discussion about a data store that would best fit our needs. We researched two caching mechanisms, Redis and Memcache. They are both similar in that they are both in memory and key-value pair storage systems. They both can be extremely faster depending on the use case.

Memcache is older and has been battled tested in many production environments. It can only store strings as data. Memcache is extremely good for storing static data the needs no further processing. The down side to memcache is it doesn’t have a native way to persist data onto the disk. Also the keys sizes is limited to 250 bytes.

Redis is a newer technology but has many features that are better than Memcache. Redis can store multiple types of data not just strings, which makes it extensible. Redis also has the ability to persist data on the disk, so if the server reboots there will still be some data in the cache. There is also a query language similar to SQL, that redis has. That allows for easy manipulation of data from the server side.
After careful consideration we decided to go with Redis because it has many features that allow it to be extensible and allow our application to scale well as more sensors get added to the system.

Another way of performing the same task would be to use a timer based system on the client side, that uses ajax calls to the back end each time it needed data. Performing the updating that way would work, but it’s not the most effective. It made more sense to put most of that responsibility in the back end of the application, instead of relying on the client to make those updates. If it was the responsibility of the client to update, each person on the website would get data at different times, varying the user experience of each user. By moving all the data update logic to the back end, each user on the site will get consistent updates from the server.

Weather Facade

A requirement of the application that it would be able to support the integration of multiple sensors. In the future there would be many sensors spread out all over campus and each of them would need to be able to have their data integrated within the system. That meant we had to achieve the extensibility quality attribute. In order to achieve that requirement, the facade pattern was implemented in order to hide to various complexity of the sensors of the system.

The facade pattern hides the complexity of the system and provides an interface that is used to access the requested data. This pattern involves a single class which provides simplified methods required by the system and delegates calls to methods of existing system classes.

Figure 4: Weather Facade
Each concrete sensor is required to have a UID variable and a getData method. The facade will generically import each sensor and make an asynchronous call to the getData method depending on the time interval in the scheduler. Once the data is received a dataCallback() method is called within the weather facade. That method checks the data and then stores it inside Redis (The system cache). Also within the scheduler there is a time interval that saves the data to the DB. After the interval is called the saveDataToDB() method is called in the weather facade. That method goes into the cache and retrieves the latest data, and stores it in the database.

Database Proxy

In order to access the database and our cache we decided to build a set of modules that would act as a middleman in retrieving data. Those modules are responsible for setting and retrieving data. The database being used in the application is called Mongodb. Weather is stored in time series so it does not make sense to use a relational database. MongoDB lets the system store data entries in timestamp order but search for entries on different metrics easier then a time-centric database would allow. The proxy decouples database implementation from the scheduler system, allowing ease of expansion or modification to the database in the future without disrupting critical sections of the code.

External Services

A requirement of the system was to be able access outside weather information from external APIs. For this project the sponsors wanted to use NWS (National Weather Service) API in order to obtain forecast and radar data. That task proved to be challenging. The NWS API is not up to date with current web standards. In fact, one of their pages for their REST API had not been updated since 2005. Also the documentation for the API is not very useful to people who have no knowledge of weather data.

In order to retrieve the five day forecast there needed to be multiple calls to the NWS REST API. The total number of calls that would be sent would be 11. One call would get the dates and the other calls would get the information for those days. To make those calls faster a JavaScript library called Async.js was used. Async.js allows for methods to be fired off in parallel. Also NWS’s REST API returns XML. The problem with that is our application understands JSON. In order to convert the XML to JSON a converter was used. After the data was parsed from XML to JSON, day objects were created and a JSON object is returned to the user.

Figure 5: Forecast JSON Data
NWS provides radar images from many different places across the United States. The issue is they have no service that will pull all the needed images for you. Essentially, NWS has a file server where those images are updated. In order to make the radar extensible a service was built in order to retrieve the HTML from the file server, parse the urls, and return an array of those images to the user.

Figure 6: Radar JSON

```
{
  - forecast: {
    - Tomorrow: {
      name: "Tomorrow",
      high: "59",
      low: "42",
      start: "2015-05-20T08:00:00-04:00",
      end: "2015-05-20T20:00:00-04:00",
      type: "day",
      weatherConditions: [ ],
      + cloud: [-],
      probabilityOfPrecipitation: "2",
      + humidity: [-],
      + minHumidity: [-],
      + maxHumidity: [-],
      relHumidity: "43",
      + precipitation: [-],
      moreInformation: "http://forecast.weather.gov/MapClick.php?textField1=41.0788&textField2=-77.63",
      + convectiveHazard: [-],
      + wind: [-]
    },
    - Friday: [-],
    - Memorial Day: [-],
    - Tuesday: [-],
    - Tonight: [-],
    - Tomorrow Night: [-],
    - Friday Night: [-],
    - Saturday Night: [-],
    - Sunday Night: [-],
    - Monday Night: [-],
    - Saturday: [-],
    - Thursday: [-],
    - Thursday Night: [-],
    + Sunday: [-]
  },
  - radar: [
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2029_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2048_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2058_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2108_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2147_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2216_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2305_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150519_2315_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150520_0004_N0R.gif",
    "http://radar.weather.gov/ridge/RadarImg/NOR/BUF/BUF_20150520_0013_N0R.gif"
  ]
}``
Configuration

Every web application needs a way to configure itself based on the environment it is being used in. In our case we had two situations, one for production and one for development. The configuration files hold important information like database and caching credentials. Since each developer's local might be set up in a different way, it makes it easier to have a set of files that will act as the glue between the application and the server.

Process and Product Metrics

The team kept time tracking in semester one and did not continue to do this for semester two. A function point analysis effort occurred in semester 1 and did not offer any meaningful results; the team decided against using it because the software of the system itself does not lend itself well to a function-based analysis (feature-light website).

One trend in the development of the project was that half of the team simply did not have the necessary skills to develop a website, and so were programming-wise unproductive until much later in the project lifetime. A simple task-completion count accountability metric was implemented and its results were skewed due to a lack of a basis for comparison between tasks. The fact that an accountability metric had to be implemented alludes to the fact that the team had troubles in getting their work done.

Product State at Time of Delivery

The product has been successfully completed. All core features and user stories are available. The system also has some extra features that were not planned as ‘core’ features - the system has a radar page, a forecast page, and an RSS feed page. The product has been successfully deployed to an RIT ITS server and is online at the time of writing.

Project Reflection

A project reflection was done in week 15 with the development team, Professor Kiser, and Bill Springer. The following text outlines what was discussed and the thoughts of those present.

● What went right?

The development team asserts that all major features were implemented and two implementation stretch goals were met. The development team delivered the core system despite project-killing circumstances by the way of scope overhaul and lack of hardware dependencies. Furthermore, the development team solicited and responded to feedback properly in an agile sense. The sponsor remarked that the site looks good as well.

● What went wrong?
Certain team members did not adapt to the web technologies needed to be used for the software implementation. This project relied heavily on hardware dependencies (weather sensors), which were never delivered to the team. The project had poor tooling for its software process methodology, and had to switch tools mid-way through the project. The team failed to turn retrospective analysis into actionable changes of the methodology.

● What would you do differently in the future?

Certain parts of this project were handled non-aggressively, which hurt the overall project effort. In the beginning of the project, the team had to email several different people to find out about hosting environments and their software restrictions; the team would be more forceful in getting answers about the hosting environments if done over again, or would enlist the department head. Bill Springer remarks that he and the other sponsor would be much more sceptical of resource acquisition for the hardware sensors and access to facilities. Coach Kiser remarks that he would spend more time reviewing the technical risks faced by the development team, so as to help them resolve any dependencies earlier.

Those issues aside, a general theme appeared in the development team’s own reflection: process. Each member of the team remarks that they would spend more time in the software process methods if done over again. The mid-term retrospective was missed for this project and everyone agrees that it should have been done.