Beatrice Evelyn Corfman
4/24/2016

RW Kern Center

Digital Dashboard Documentation

The Kern Center Digital Dashboard is implemented via Ruby on Rails, with a standard Model-View-Controller architecture. The initial implementation was done in Fall 2015 as a part of *CS-0291: Software Engineering* (Beatrice Corfman, Kyoko Sano, Mike Dean, Nicholas Lee, and Ella Holmes). Development was continued by Beatrice Corfman in Spring 2016, as her Division III project. It can be found at http://kern-dashboard.hampshire.edu

This readme covers design goals, system architecture, database structure, shortcomings, and ways to expand the system. It should be referenced when trying to troubleshoot or expand the Rails end of the dashboard.

It does not cover general Rails questions except where directly relevant: Rails guides can be found at http://guides.rubyonrails.org/

Design Goals

The Digital Dashboard's primary purpose is to teach visitors to the Kern Center about the building itself, the Living Building Challenge, and to display information about water and electricity usage/generation within the building.

As a front-facing Hampshire College website, openly displayed in what is to be the student and visitor hub on campus, it's exceedingly important that the look and feel of the website match Hampshire's branding. Information on visual identity can be found on the Communications Office page on Hampshire's website – modifying the look of the page as Hampshire's visual identity changes or evolves will be very important.

Although perhaps counter-intuitive and against the Communications Office web publishing guidelines, it's actually very important that the dashboard contain **no external links whatsoever**. This is because it will be on display on a large screen in the Kern Center, which should, unless specifically desired otherwise, only display the dashboard. Putting in external links means someone could navigate to other websites and then leave them up on the screen, or navigate away from the page and be unsure how to go back. While it would theoretically be nice to have some external links in some places, the end result would be too destructive to the dashboard displayed in the Kern Center lobby.

System Architecture

The dashboard is hosted, as of this writing, on a machine named Eurus (192.33.12.92), under care of the Hampshire College IT Department. It uses Apache and Phusion Passenger, and runs on Ruby 2.3.0p0 and Rails 4.2.0.
The rails application itself can be found in /var/www/kerncenter/code, and "kernuser" has permissions to write to the folder (the password is not being recorded here, both for security reasons, and because it is liable to change more than other things that could be written in this document. Hampshire IT should be able to give it to you if it's appropriate for you to have it).
The MySQL server that the application uses is also hosted on this machine, and is detailed more explicitly in the Database Architecture section. Source code is currently versioned via https://github.com/notthatstraight/kerncenter.

The Rails app is structured fairly conventionally, with extra care taken to abstract large pieces of code into helper methods and partials. Most of what is run to display the website is in /app, database configuration and migrations are in /db, and other miscellaneous things (excluding the gemfile in the home directory) are in /config.
The application is currently split into four main workflows: Application, Dash, Resource, and Programs.

Application

This is the overall workflow for the entire application. It contains things that affect the whole site, such as general (S)CSS, navigation, headers, etc. /app/views/layouts/application.html.erb is also part of the Application workflow, and is where any HTML and includes you want affecting the entire site should go.

Dash

This is the homepage workflow. There's very little done here; most things that affect the homepage affect the entire site, and so go in Application instead.

Resource

This is where most of the current heavy lifting in the app currently lives. It handles database queries and data transformations for displaying sensor data by resource type. There are two pages currently under its purview, water and electricity. They both share large amounts of the same code and differ only in which resource is displayed; they've been split up into different pages to more easily enable page detection for the navbar. All of the database calls and transformations have been abstracted into /app/helpers/resource_helper.rb, so that splitting up the pages doesn't duplicate code unnecessarily.

Programs

This workflow is currently a largely empty shell. It's intended to hold similar functionality to the Resource workflow, but modified to pull by each "Suite" of sensors

Helpers

A large amount of the processing done do serve pages on the dashboard is done inside of *helpers* (/app/helpers). These are files that contain various different functions, which can be called from views in the same naming scheme (e.g., resource_helper.rb is available from anything in the /app/views/resource folder). It's important to note that helpers are only visible from views – they are **not accessible from controllers**. Their purpose is to strip code from views so that views can be primarily/entirely display information.

There are three helpers which currently contain functions:

application_helper.rb contains two functions.
The first, granularity, takes two inputs: a start date and a stop date. It then calculates and returns a list of pairs of dates, which divides the passed date range into a list of discrete chunks. These chunks can then be used to display a graph of data from inside the given date range. For example, if you gave it 3-21-2016 and 3-27-2016, it will return:
[[3-21-2016, 3-22-2016], [3-22-2016, 3-23-2016], [3-23-2016, 3-24-2016], [3-24-2016, 3-25-2016], [3-25-2016, 3-26-2016], [3-26-2016, 3-27-2016], [3-27-2016, 3-28-2016]]
n.b., the last date is outside the range of the dates input. This is simply because the function that sums the data for a date chunk together is inclusive on the lower bound and exclusive on the upper bound (That function is a method in /app/models/sensor.rb):

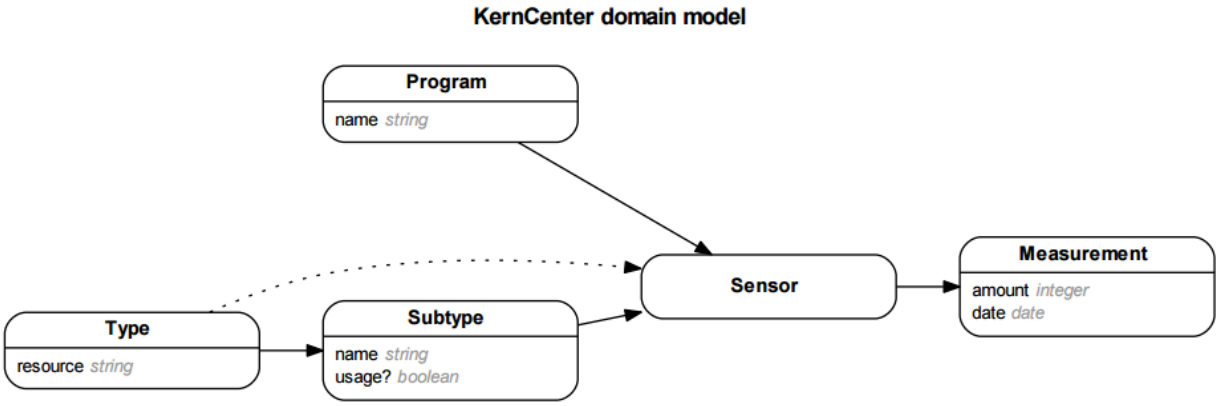
```
def amounts (start, stop)
   self.measurements.
   where("date >= ?", start).where("date < ?", stop).
   map {|m| m.amount}
end
```

The second function is makeLink. This is intended for making navigation links; it takes a url, text, and a color, and returns a circle link of the given color (although the color needs to be included in nav.scss for it to work; otherwise, the circle will just be black). It also checks whether the link it's been asked to make is for the current page; if it is, instead of returning a link, it returns a visually similar (but noticeably distinct) non-link.

chart_helper.rb contains every function that directly uses the lazy_high_charts gem to make Highcharts graphs. Anything that utilizes lazy_high_charts functionality should go in here (or in another, more specific helper, which should then be included in chart_helper). chart_helper should be included in any helper associated with a view that needs to display charts (such as resource_helper and programs_helper).

resource_helper.rb contains the function getResourceVars (and includes chart_helper). getResourceVars takes the name of a resource, and uses the :start and :stop date params passed to the view, to return an array of all of the data needed to pass to a chart_helper function, in order. n.b., this is an array: to use it in a chart_helper function, you need to unpack it first using the * operator (*array returns the items of the array, in order, not in an array).

Database Structure



Generated using rails-erb

The database for the Kern Center Dashboard is divided into five tables, as shown in the diagram above. More explicitly:

Measurements is every single measurement associated with a sensor at the Kern Center. Every measurement is associated with a particular sensor (via a sensor_id), has a flat measurement (the cumulative total on that sensor for a given day), and a date.

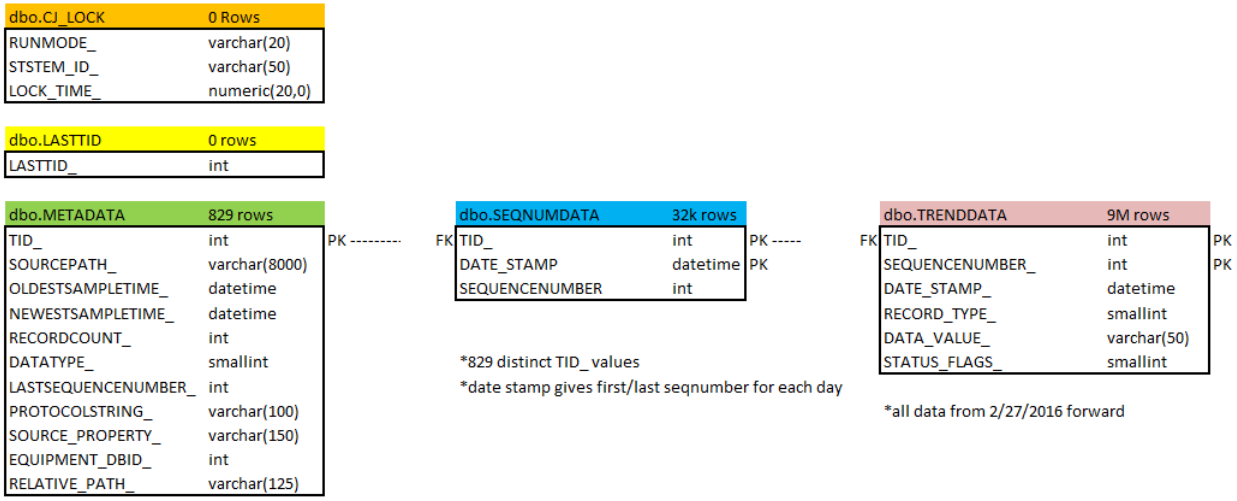
Sensors is every sensor associated with the Kern Center. Each sensor is associated with a program (program_id) and a subtype (subtype_id). They're also all associated with a type, indirectly, through subtype – this isn't represented in the database, but there's an explicit statement in the Sensor model (/app/models)

Programs is a table of all of the "suites" of sensors in the Kern Center. This is the finest granularity of area-based measurement: there are no sensors for individual rooms, just sensors for areas of the building. Every program has a name, which is a string intended for display use on the dashboard (e.g., "Lobby & Café").

Types is every overarching type of resource. There aren't many of these – currently, it's only water and electricity, and that's unlikely to change in the future. These simply consist of a name, used both for lookup and display. (n.b., these should be lowercase. Resource names could theoretically be used in the middle of a sentence, and they can simply be converted to title case where needed with a string transformation. While they're used more frequently in title case, it's more clear to leave them lowercase and then transform them).

Subtypes is all of the different types of a specific resource there are. Each subtype is associated with a type (type_id). Subtypes have a name, intended for display purposes (e.g., "plug" or "mechanical"), and also a Boolean value that notes whether they represent resource generation, or resource use (false and true, respectively).

Automated Logic Database



The database that the dashboard itself uses is not actually the origin point of the sensor data. Automated Logic, the company that installed the sensors in the Kern Center, has their own database, which collects data from every building on Hampshire campus.
The dashboard uses a different database for a few reasons; mainly, to keep it as self-contained as possible. A priority of design should always be that the entire system can be slotted and/or changed without modifying anything external.
Additionally, it's simply not a good format for the application, and contains data that the dashboard has no use for – among other things, data on buildings that aren't the Kern Center. Even more than that, the Automated Logic database wipes data more than 30 days old, and the dashboard needs to have more historical data than that.

The data from the Automated Logic database is pulled into the dashboard database by querying use_prev_day_tn shortly after midnight, extracting the data into a csv, and putting that csv on Eurus. A batch process can then be run that puts the data in the csv file into the database.

Data should generally not be written into the database any other way, to avoid unexpected performance.

Shortcomings

There are a number of features that haven't been implemented in the dashboard rails application. Some of these were simply cut for lack of time; these are covered in the Expansion section. Others were unable to be implemented due to hardware constraints, and are detailed here.

On any college campus, and at Hampshire College especially, offices are prone to move around and change which building/rooms they're in. It would therefore be ideal to be able to track when any particular office inhabited any particular space.
Initially, there were two additional tables in the dashboard database to allow for this: rooms, and room_program_time_mappings. Instead of being associated with programs, sensors were associated with rooms. Rooms and programs were then mapped through the room_program_time_mappings table – each row had a room_id, a program_id, a start, and a stop. Programs could thus be mapped to rooms flexibly, and could change which rooms they occupied over time. Any query that was program-specific could be run through the room_program_time_mappings table, and then pull data for the sensors associated with the rooms associated with the program during the time overlap between the query dates and the occupancy dates.

Unfortunately, it's unwieldy and unrealistic to put sensors in every room of a building. Sensors are instead put in panels, which serve entire areas of the building, called "suites." There's a Lobby and Café suite, an Admissions 1st Floor suite, etc.
Since suites are the finest area-based granularity at which we can measure, the gain from being able to track occupancy changes over time is minimized, and the additional query load and structural complexity from organizing the database becomes a much more relevant concern. The rooms and room_program_time_mappings tables were thus dropped.

Expansion

There are a number of directions in which this system could be expanded. The most glaring is the current lack of a working Programs workflow, cut due to time constraints.
The Programs workflow primarily needs code written to query the database, similar to the code in the Resources workflow – though that would need to be modified somewhat. Additional routing would also need to be done.
Additionally, the 'amounts' method in /app/models/sensor.rb should be updated to include an optional programs argument:

```
def amounts (start, stop)
   self.measurements.
   where("date >= ?", start).where("date < ?", stop).
   map {|m| m.amount}
end
```

More ambitiously, it would be fairly interesting to be able to compare two programs in the same graph. That shouldn't take a lot more work on the database-querying end, but routing well might become a bit complicated.

Finally, a much simpler modification would be providing a date-selection slider. Currently, there are buttons which allow you to pick a few date ranges (past week, 30 days, and twelve months), but the back-end and routing are capable of handling any two ordered, different days. A slider with start and stop values could be implemented with jquery-ui-rails (a Rails gem that allows you to work with JQuery UI), and used instead of, or in addition to, the buttons It could be made particularly clean by setting its default values to the start/stop values passed to the view, as then the default selection would match the dates of the page currently being viewed.
While similar date-selection capabilities could be offered through the use of text fields, a slider would be much more touch-screen friendly.

Adding controllers

Controllers can be added from a rails console using the command "rails g controllername actionname1 actionname2…" (the actions are optional but recommended). Controller actions can be views (explained below), or can contain functionality such as POST actions.

Adding views

Views can be added as .html or .html.erb files, in the folder in /app/views that corresponds with the controller and workflow you want the view associated with. All views need to have an action of the same name in the controller of the folder they're in – if you put index.html.erb in /app/views/dash, there needs to be an index action in /app/controllers/dash_controller.rb. This action doesn't have to necessarily do anything, but it does have to be there, because rails routes to controller actions, not directly to views.

Views are generally .html.erb files – that means they're embedded ruby files, that are evaluated down to HTML files. Ruby is included in a .html.erb file by enclosing it in either <% these %> or <%= these %>. <%= puts the value of something on the page, and <% is other code that doesn't get printed to the page.

Additionally, you'll need to add routing for the new page to /config/routing.yml, or rails won't know which url(s) should go to the view in question. There's a lot of routing functionality available, so it'd be best to just look at the documentation for it.

In addition to plain views, rails supports partials. These are placed in /app/views/shared, and have a filename that starts with an underscore (_). They contain pieces of pages that can be included like so: <%= render :partial => "shared/header" %>

Modifying the database

Database modifications in rails are done via migrations. Migrations can be made from a rails console using the command "rails g migration migrationname"; this will create a properly-timestamped migration file in /db/migrations, which can then be edited to achieve the desired effect on the database (migration file functionality can be found on the rails website).
Running rake db:migrate from a rails console will run any migrations that haven't been applied to the database, in order.

Editing (S)CSS

Most of the CSS for the dashboard is actually in the form of .scss files, which is the Sass file format. CSS can be directly written into a Sass file, but you have access to additional functionality. The entire file gets compiled down into a .css file.