We’re working full time this summer alongside 3 UCOSP (project course) students (2 from Waterloo: Mark Rada & Su Zhang, 1 from UofT: Angelo Maralit)

Our supervisors:
Karen: heads project, which has been in development/production for a few years (from 2009-ish)
David: PhD student/lecturer, used MarkUs and is now supervising the project
What is MarkUs?

“Who has(n’t) used MarkUs?”
An open-source tool which allows users to easily manage, grade, and submit assignments within a web app.
“Who has(n’t) used MarkUs?”
An open-source tool which allows users to easily manage, grade, and submit assignments within a web app.
“Who has(n’t) used MarkUs?”
An open-source tool which allows users to easily manage, grade, and submit assignments within a web app.
“Who has(n’t) used MarkUs?”
An open-source tool which allows users to easily manage, grade, and submit assignments within a web app.
Workflow goes like this:
Instructors manage all other users, but more importantly they create assignments.
Students (hopefully) complete assignments and submit them for grading.
Graders, usually teaching assistants, grade the assignments. They can fill out the rubric, add comments, and annotate files. Instructor releases them for students to view, and can also ask for a remark.
Where is it used?

- University of Toronto St. George and Mississauga
- University of Waterloo
- École Centrale de Nantes (ECN), France

In addition to being used at UoT, MarkUs is in use for Waterloo's first year CS course as well as in France, at ECN. MarkUs is fully internationalized, thanks to translations from our past French and Portuguese-speaking developers!
How is MarkUs built?

• Ruby on Rails framework
• JavaScript
• Subversion
• PostgreSQL (or MySQL)
• Apache

Our techstack is somewhat vanilla, viz.
Backend: Ruby on Rails
Front end: JavaScript
File manager/version control: SVN
Database: choice between Postgres/MySQL
Server: Apache
Today’s talk

• Design: SASS
• Performance: React.js, optimization
• Testing: RSpec

We’ve been busy squashing bugs this summer, along with other things. Today, we’ll be talking about design-related things – for the front and back ends, and software design before the coding even begins!
A website can basically be seen as 3 parts [building a house metaphor]: HTML (structure: foundation) + CSS (style: paint/furniture) + JS (functionality: appliances)

We wanted (re: needed) to refresh MarkUs’s 5 year old design, so we took this opportunity to switch to SASS.

With SASS, it allows for easier maintenance, a more modular style sheet system, and it makes it easier for working on a more responsive (mobile) design.

*Note: we’re using SCSS for MarkUs.*
We leverage SASS to easily make things responsive and modular.
Example: some simplified code from MarkUs.

This is a partial file, demonstrating SASS’s variables, maps, and mixins.
This “breakpoint” mixin will be used for easily adding media queries for making MarkUs responsive.
Main file: we’re importing the partial from before and using its variables and mixins. We can also see an example of inheritance (the @extend call), and SASS functions (lighten).
Plus, the nesting demonstrated here is very useful for keeping everything clean.
Compiled: pretty trivial in this case. SASS can also be useful for things like dealing with all the vendor prefixes, along with using variables and such for modularity that wasn’t possible with vanilla CSS.

```css
main.css
.menu { background: #89b1dd; }

.mobile_menu {
    background: #89b1dd;
    display: none;

    @media (max-width: 745px) {
        display: block;
    }
}

.mobile_menu a {
    color: #bce4ff;
}
```
You can see a demo version of MarkUs running at www.demo.markusproject.org. See www.markusproject.org for more information.
The style guide/UI toolkit is available at www.echeung.me/markus-style (but it should be on the official website at some point).
Vagrant is a virtual environment manager that makes it really easy to create and configure developer environments. MarkUs has a lot of dependencies and the installation docs constantly become out of date and need to be updated. Now that MarkUs uses Vagrant, we only need to run 2 commands to get a configured virtual environment. Additionally, if everyone on the team is using Vagrant, no bugs will crop up due to one person having a slightly different configuration, since everyone is running identical systems.

- MarkUs has a substantial amount of dependencies
  - Ruby
    - Rails + many other gems
  - PostgreSQL
  - Subversion + ruby-svn bindings
VAGRANT

• Virtual environment manager
• Supports most virtualization software
  • VirtualBox (default)
  • VMWare
  • Docker

VM vs Vagrant
BUT, you can do this with a traditional virtual environment like VirtualBox or VMWare, so why use Vagrant?
Vagrant removes the hassle of setting up, configuring, and distributing environments.
Example:
Top - Vagrantfile. "markusproject/debian" is the identifier for the virtual env I've set up on Vagrant Cloud.
"forwarded_port" means when I run a server in the guest VM on port 3000 I can connect to it through localhost on 42069.
Bottom - usage example.
Note that you can do all this in a traditional virtualizer, but you would have to manually do the setting up and configuration. Vagrant just makes it easier to do and thus less likely to screw up.
One easy-to-fix bottleneck for performance comes from the N+1 problem. This problem arises due to the object relational mapper in rails called ActiveRecord, which converts database results into ruby objects. It makes it very nice to program, but it doesn't necessarily know the optimal calls to make. The cause for the n+1 problem is that AR lazily loads associations, which 90% of the time you want, because you don't want to load an assignment and at the same time get every single user associated with that assignment if you just want to know about the assignment. But for the other 10 percent of the time it can be a major bottleneck. as an example for the problem, this is some code from MarkUs that gets all users that have been accepted into a group. At this point in time (membership.user) ActiveRecord can't see that we're actually getting all the memberships, so it does the dumb thing and loads them one by one. N students equals N calls, plus one for the original StudentMembership call. This obviously gets slow when you have a lot of users, so let's try to avoid it.
It turns out that we can actually reduce this linear time solution to constant time by doing something called eager loading. Eager loading is when you manually tell rails that you also want to load this association along with the original one, so you don’t waste time looping through each membership. So this makes MarkUs faster by trimming a potential 1000 database calls down to just 2.
Here is a screenshot of one page that was plagued with this problem. It's the admin view for assigning and unassigning TAs to groups. It sometimes took several minutes to randomly assign every TA to every group because of the n+1 problem.
But now it's fixed, and you can see the new page render times (green is without, blue is with). This is for only about 100 students, and I bet the differences would be even more drastic for more.
But that doesn't solve all our problems. There are several problems with the front-end of MarkUs, as in the javascript used to asynchronously make requests and update the page without refreshing it.

First of all is that it's messy. There are a lot of identical one-liner files, onClick and onKeyUp handlers that just make it hard to keep track of state. It's also slow for a similar reason as the n+1 problem in that every table row for every table is being rendered one at a time in the backend, which is also problematic for other reasons. So I decided to use this javascript framework called React, which is in use at Facebook and Instagram, to re-implement the tables used in MarkUs.
React is a functional way to build user interfaces. Generally the way to use it is to build reusable components which correspond to html, and then render one of the components onto the DOM. This is some example code for the StudentsTable in MarkUs. I want to draw your attention to the render function, which defines which columns should be in the table and whether they should be sortable and searchable, as well as filters that define a function that a piece of data must pass or fail to be either considered in the filter or not. Then we render out a StudentsActionBox which takes care of performing actions on the selected students which are like placing them in a section or giving them grace credits. We also render out another component, which is a generic table that I've written with support for sortable and searchable columns. So then we let these two components, which may or may not have their own subcomponents, handle the rest of the view logic.
So this is what it ends up looking like.
Here is the component layout for a more complicated view, which assigns students to groups.

As a result, bugs are much easier to reason about. Data flows mostly only one way, since React is functional, so if something is off you can just check the data you are getting at each level.

Also this modular design encourages code reuse which is always good.
And this is what it comes out to.
Creating a testing model for markus using RSpec. Most used in ruby community, making it easier for new developers. Easier to understand tests can be useful to new developers as reference. To act as documentation for software.

**RSpec**

- Switching testing frameworks
- More precisely, writing a testing guide for MarkUs
- Why switch to RSpec?
  - Most popular testing framework in Ruby community, which means more resources for new developers
  - Resemblance to natural language allows for easy understanding of tests, and a sort of documentation for software
When upgrading, want confidence nothing broke.
Testing became less frequent due to increase in complexity.

Best way to have a strong set of tests is have developers test their code.
Looks trivial, but when user supplied need to verify. Example of it being readable (NLS)
Not the actual conversion that’s difficult, but understanding if that test is worth
converting.
Su and I were clashing over small details. So many ways to do the same thing, must
stay simple.

Almost done testing guide, will start something else when completed.

RSpec

• Challenges
  • Test conversions, is the test worth converting?
  • Design decision discrepancies, what criteria should our tests have
    (ie. readability, elegance, etc)

• Status
  • Finished writing the first draft of the guide
We’re trying to offer Git as an option instead of just SVN. jQuery is now the industry standard, so we’re transitioning the old Prototype code.

Where are we headed?

• Milestone 1.2:
  • Git backend
  • Transition from Prototype to jQuery
  • Upgrade to Ruby 2.x and Rails 4.x
MarkUs is open source (on GitHub at www.github.com/MarkUsProject/): feel free to report bugs, or even submit fixes!