Final Project Report

CPSC 301
2012.04.11

Team Pleiades

Stephen Huang
Evan Ranshaw
Kent Williams-King
Mike Wing
Jeff Wintersinger

Project Evolution

While developing Pynx, our most surprising discovery was just how much interpersonal communication was required to coordinate development efforts, as will be discussed in the subsequent section. This in turn reduced the amount of time team members had to do work reflected directly in the end product, inhibiting our ability to deliver on the full set of features we planned for our first iteration. The harried pace and short development cycle meant that little formal prioritizing of features was required — the fundamental aspects of the system on which other features depended were developed first, with peripheral functionality largely ignored until necessary. As a result, several features both large and small that we had previously seen as integral to the system were dropped, including tagging of discussion posts, pagination of discussion topics, and notification of new memos. This last feature's necessity was obviated as a result of our narrowing scope — while we had initially planned to show the user a "dashboard" overview of relevant activity, we decided that the amount of information presented by the system's current incarnation is sufficiently humble as to render this function unnecessary. Instead, we directed the user to the memos listing upon logging in, removing the need for an explicit memo notification system.

Just as our feature set narrowed, so too did our test plan. At development's outset, we planned to perform extensive automated testing on the server and manual (though regular and structured) testing on the client. At first, we even toyed with the notion of investigating Selenium for automated client-side testing, but quickly discarded this idea once the extreme time pressures of the development cycle became apparent. Though our intent was to perform scheduled manual testing of the client, this too slipped — instead, we found our meetings frequent enough and our feature set sufficiently paltry that working through newly added functionality as a group was sufficient. Additionally, though we at first endeavored to perform automated performance testing of the server, lack of time and logistical concerns related to establishing acceptable performance metrics prevented our doing so.

Though we did experience success with our server-side unit testing, our coverage was less considerable than planned. We ended up with tests focused on memo and discussion functionality, largely ignoring authentication and the administration panel. This shortcoming resulted from the tremendous time investment required to develop our own testing infrastructure. While "full-stack" web development frameworks such as Ruby on Rails and Django ship with tightly integrated unit-testing and fixture-loading functions, Node is much more agnostic as to technology choice. Thus, we had to seek out a test framework; even once we settled on Mocha, we had to work to integrate it with the rest of the project.

Mocha's greatest shortcoming was its lack of support for loading fixtures as a means of putting the database into a known state before undertaking tests. This in turn forced us to develop our own fixture-loading framework, which imposed

challenges of its own. Our initial implementation had the testing framework interfacing directly with the database to load fixture data, which resulted in errors occurring at irregular intervals relating to database file write lock contention between the test and server processes. This was all the more aggravating given the assurances provided by the SQLite documentation that multiple processes could readily utilize the database at the same time; we suspect that the locking behaviour of our chosen Node.js SQLite bindings may be at fault, with the library being all too eager to obtain a write lock for the database. After futile hours of debugging, we rewrote our fixture framework to load fixtures by issuing an HTTP request to the server, which would then load the fixtures itself, eliminating issues related to multiple processes interfacing directly with the database. (The server's fixture-loading functionality does, of course, present a security risk, and is thus active only in the test environment.) In this fashion, the immaturity of the Node ecosystem stymied our testing efforts — some team members spent an undue amount of time building essential "plumbing" functionality such as fixture loading that is provided out-of-the-box in other frameworks, while the frequent interface changes necessitated by underlying technological challenges hampered other members' ability to write tests.

Project Lessons

The most surprising lesson imparted by this project was just how much time is required to coordinate efforts of multiple team members. As none of us had before worked with more than one person, the difficulty of reconciling the products of five people's labours came as a revelation. Compounding the problem was the divergent skill sets of each member — while some had past experience with web development, others were entirely new to the field. Though our summed development hours totalled only 75 hours, at the low end of our estimated 74 to 122 hour range, the time we spent conversing with one another added considerably to this total. Through each week of the project, we met at least three times per week; in several of those weeks, we came together every day, pushing the time we spent meeting in person to perhaps 30 or 35 hours across the project's length. In addition to this extensive in-person time, we often collaborated online over the weekends, making use of IRC and a web-based discussion group. Most surprising was just how much higher interpersonal bandwidth is in person rather than online — explaining a problem or demonstrating a technique is much more easily accomplished in the flesh, for it allows both parties to hone in more quickly on a mutual understanding. Extensive though our meeting time was, it was entirely worthwhile. Time invested in planning ensured that our end product was constructed on an extensible base, while our discussion efforts ensured that all team members could contribute to the project despite our differing amounts of web development experience.

For project members without past experience in web development, the gamut of technologies they had to learn was overwhelming. Though the course's labs

provided a grounding in Node and JavaScript, building a full-fledged application required that team members be competent with a bevy of additional technologies, ranging from the SQLite database library, to the Express web framework, to the Mocha testing framework, to the Backbone.js "thick-client" framework, to jQuery. While more seasoned team members worked frantically to establish a technical base, those unfamiliar with these technologies scrambled to familiarize themselves with the technologies' essentials, often feeling lost and unsure of what work to pursue. A more effective organizational strategy might well have been to appoint a single leader responsible for ensuring that every team member could contribute effectively at all points of the project, rather than relying on a bottom-up, self-organizing strategy.

The project's most frustrating aspect was the immaturity of the Node.js ecosystem. While full-stack frameworks such as Rails and Django impose their preferences on you, forcing you to conform to a set of "best practices" established by developers well-versed in the field, Node left us largely to our own devices. Despite some team members having previous web development experience, we still found ourselves floundering during the project's early days. We initially resolved to write our own server from the ground up, basing it on the past structure of one member's lab assignment; the substantial amount of work required for such elemental tasks as cookie parsing, however, soon forced us to consider a more robust solution. Though we promptly switched to the Express framework, this change obsoleted the team's understanding of the server, requiring each member to learn anew. The afore-elucidated troubles relating to fixture development serve as a splendid example of the frustrations engendered by Node's minimalist philosophy; had management not imposed the Node.js choice on us, we likely would have sought out a framework offering a richer set of building blocks.

Our team's most ambitious decision was to pursue a thick-client approach. We deemed this methodology preferable to the traditional thin-client philosophy prevalent in web applications, in which the server provides fully-rendered HTML pages and little code executes on the client, as it would enable a richer, lower-latency interface. Also attractive was the potential of this approach to simplify our server-side functionality — given Node's minimal tool set, any means of reducing the server's complexity seemed ideal. This plan of attack worked surprisingly well. Our finished server provided only a single static HTML page used to "bootstrap" the application, and all dynamic data was returned in JSON format. The server's complexity was indeed minimized, allowing us to ignore issues such as choice of template libraries, given that we never served anything not in JSON format. Additionally, the client-side interface became exceedingly responsive, as no full page reloads are required at any point of our application's use. Despite this, convention-breaking issues with which early rich application had to content, such as the lack of unique URLs associated with each page and the crippling of the browser's back and forward buttons, are overcome through our use of HTML5. Of course, these benefits came at the cost of increasing the amount of background knowledge required by team members before they could contribute to development. However, given the end product's effectiveness, we

believe this trade-off was worthwhile.

When we reflect on the sum total of our experiences through this project, most interesting are not the lessons learned regarding technology, such as the worth of pursuing rich-client interfaces or the frustration of writing one's own fixture library, but those that relate to the interactions of individuals composing a team. Though we are all competent as individual developers, integrating our efforts to produce cohesive progress as a team presented novel challenges. The difficulties of interpersonal communication are never more apparent than when we reflect on the amount of time required to coordinate our efforts – in our final analysis, we spent nearly half as much time simply designing and discussing the system as we did writing it. As little real-world software is written in isolation, however, these lessons, no matter how hard-won, will serve us well in our eventual careers.

Iteration plan retrospective

Category	Feature	Task	People	Timeframe	Time Taken
Architecture	Front-end architecture	Implement MVC framework	Kent/Jeff	2-4 hours	10 hours
		Implement templating framework	Kent/Jeff	2-3 hours	3 hours
		Implement client-side integration test protocol	Kent/Jeff	2-4 hours	0 hours
		Implement client-side library for working with DOM	Kent/Jeff	1-2 hours	1 hour
		Construct overall framework for loading sub-pages in single containing primary page	Kent/Jeff/Stephen	3-6 hours	5 hours
	Back-end architecture	Implement generalized JSON response structure	Evan	1-2 hours	2 hours
		Implement server-side unit test framework	Evan	2-3 hours	15 hours
		Integrate performance testing support into unit tests	Evan/Michael	3-4 hours	0 hours
Memos	Display new memo	Implement memo pane UI	Michael	3-4 hours	0 hours
		Track read status of memo	Jeff	2-3 hours	2 hours
		Notify user of newly posted memo	Evan	1-2 hours	0 hours
		Allow user to comment on memo	Stephen	4-6 hours	0 hours
	Display memo archive	List all posted memos	Michael	2-3 hours	2 hours
		Paginate memo list	Michael	1-2 hours	0 hours
	Allow admin to post new memo	Implement admin memo-posting UI	Kent/Michael/Stephen	2-3 hours	3 hours
		Implement backend memo store	Evan	1-2 hours	1 hour
Discussions	Create new discussion	Implement UI to enter topic and initial post	Stephen	4-5 hours	6 hours
		Implement backend to write discussion to data store	Evan	1-2 hours	3 hours
	List discussions	List all discussions	Stephen	2-3 hours	3 hours
		List all posts in given discussion	Jeff	2-4 hours	5 hours
		Paginate posts in given discussion	Jeff	1-2 hours	0 hours
		Sort discussions by last post time	Stephen	1-2 hours	1 hour
		Sort discussions by popularity	Stephen	3-5 hours	0 hours
		Track read status of discussions	Michael	2-3 hours	0 hours
	Manipulate discussions	Post reply to existing discussion	Stephen	2-3 hours	1 hour
		Allow user to edit/delete own post	Kent	0.5-1 hours	0 hours
		Allow admin to edit/delete any post	Kent	0.5-1 hours	0 hours
		Allow admin to delete whole discussion	Kent	0.5-1 hours	0 hours
	Tag discussions	Allow discussion poster to add one or more tags to discussion	Michael	3-4 hours	0 hours
		Allow discussion poster to edit tags associated with discussion	Michael	0.5-1 hours	0 hours
		Allow admin to edit tags associated with any discussion	Michael	0.5-1 hours	0 hours
		Allow user to display only discussions bearing certain tag	Evan	1-2 hours	0 hours
		Display all tags used in system in separate tag section	Evan	2-4 hours	0 hours
Authentication	Log in and log out	Implement session-handling support	Evan	3-5 hours	6 hours
		Authenticate user via name and password	Evan	1-2 hours	2 hours
		Log out by destroying current session	Evan	0.5-1 hours	2 hours
		Implement anti-brute-force attack prevention	Michael	5-7 hours	0 hours
Admin	Manipulate user accounts	Create new user account	Kent	3-4 hours	2 hours
		Lock existing user account	Michael	1-2 hours	0 hours
		Change user's password	Jeff	1-2 hours	0 hours
			Total	74-122 hours	75 hours