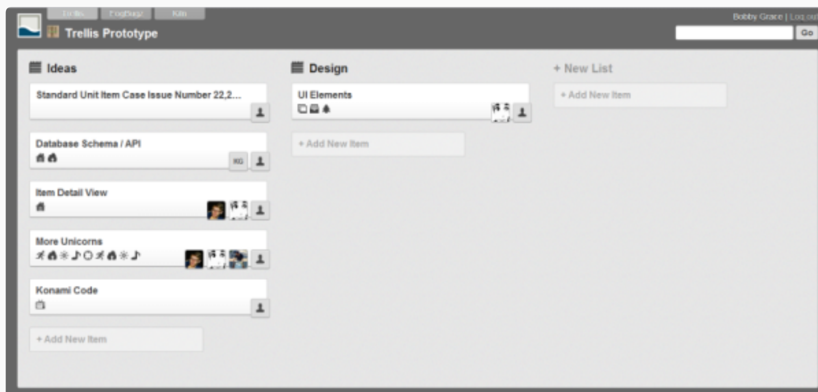
Articles and interviews on
Business, Design, Programming, Support and other Resources

The Trello Tech Stack

By Brett Kiefer on Fog Creek Product Trello

Trello started as an HTML mockup that Justin and Bobby, the Trello design team, put together in a week. I was floored by how cool it looked and felt. Since Daniel and I joined the project to prototype and build Trello, the challenge for the team has been to keep the snappy feeling of the initial mockups while creating a solid server and a maintainable client.



The Initial Trello Mockup

That led us toward a single-page app that would generate its UI on the client and accept data updates from a push channel. This is pretty far from any of the work we've done before at Fog Creek, so from a technical perspective Trello has been an adventure.

Initially, we were wondering how interesting and far-out the stack could be before management got nervous, but our concerns were addressed in an early meeting with Joel, when he said "Use things that are going to work great in two years."

So we did. We have consistently opted for promising (and often troublesome) new technologies that would deliver an awesome experience over more mature alternatives. We're about a year in, and it's been a lot of fun.

CoffeeScript

Trello started out as a pure JavaScript project on both client and server, and stayed that way until May, when we experimentally ported a couple of files to CoffeeScript to see how we liked it. We loved it, and soon converted the rest of the code over and started coding CoffeeScript exclusively.

CoffeeScript is a language that compiles to *readable* JavaScript. It existed when we started Trello, but I was worried about the added complexity of having to debug compiled code rather than directly debug the source. When we tried, it, though, the conversion was so clean that mapping the target code to the source when debugging in Chrome required little mental effort, and the gains in code brevity and readability from CoffeeScript were obvious and compelling.

JavaScript is a really cool language. Well-written CoffeeScript smooths out and shortens JavaScript, while maintaining the same semantics, and does not introduce a substantial debugging indirection problem.

The Client

- Backbone.js (client-side MVC)
- HTML5 pushState
- Mustache (templating language)

The Trello servers serve virtually no HTML. In fact, they don't serve much client-side code at all. A Trello page is a thin (2k) shell that pulls down the Trello client-side app in the form of a single minified and compressed JS file (including our third-party libraries and our compiled CoffeeScript and Mustache templates) and a CSS file (compiled from our LESS source and including inlined images). All of that comes in under 250k, and we serve it from Amazon's CloudFront CDN, so we get very low-latency loads in most locations. In reasonably high-bandwidth cases, we have the app up and running in the browser window in about half a second. After that, we have the benefit of caching, so subsequent visits to Trello can skip that part.

In parallel, we kick off an AJAX data load for the first page's data content and try to establish a WebSocket connection to the server.

BACKBONE.JS

When the data request returns, Backbone.js gets busy. The idea with Backbone is that we render each Model that comes down from the server with a View, and then Backbone provides an easy way to:

1. Watch for DOM events within the HTML generated by the View and tie those to methods on the corresponding Model, which re-syncs with the server
2. Watch the model for changes, and re-render the model's HTML block to reflect them

Neat! Using that general approach, we get a fairly regular, comprehensible, and maintainable client. We custom-built a client-side Model cache to handle updates and simplify client-side Model reuse.

PUSHSTATE

Now that we have the entire client app loaded in the browser window, we don't want to waste any time with page transitions. We use HTML5 pushState for moving between pages; that way we can give proper and consistent links in the location bar, and just load data and hand off to the appropriate Backbone-based controller on transition.

MUSTACHE

We use Mustache, a logic-less templating language, to represent our models as HTML. While 'harnessing the full power of [INSERT YOUR FAVORITE LANGUAGE HERE] in your templates' sounds like a good idea, it seems that in practice it requires a lot of developer discipline to maintain comprehensible code. We've been very happy with the 'less is more' approach of Mustache, which allows us to re-use template code without encouraging us to mingle it with our client logic and make a mess of things.

Pushing and Polling

Realtime updates are not a new thing, but they're an important part of making a collaborative tool, so we have spent some time on that layer of Trello.

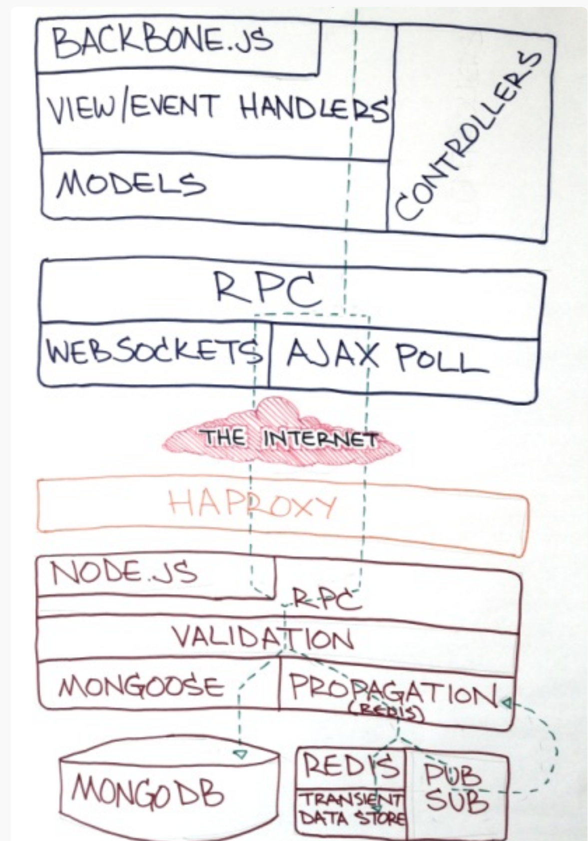
SOCKET.IO AND WEBSOCKETS

Where we have browser support (recent Chrome, Firefox, and Safari), we make a WebSocket connection so that the server can push changes made by other people down to browsers listening on the appropriate channels. We use a modified version* of the Socket.io client and server libraries that allows us to keep many thousands of open WebSockets on each of our servers at very little cost in terms of CPU or memory usage. So when anything happens to a board you're watching, that action is published to our server processes and propagated to your watching browser with very minimal latency, usually well under a second.

AJAX POLLING

It ain't fancy, but it works.

When the client browser doesn't support WebSockets (I'm lookin' at you, Internet Explorer), we just make tiny AJAX requests for updates every couple of seconds while a user is active, and back off to polling every ten seconds when the user goes idle. Because our server setup allows us to serve HTTPS requests with very little overhead and keep TCP connections open, we can afford to provide a decent experience over plain polling when necessary.



Early Architecture Drawing

We tried Comet, via the downlevel transports for Socket.io, and all of them were (at the time) shaky in one way or another. Also, Comet and WebSockets seemed to be a risky basis for a major feature of the app, and we wanted to be able to fall back on the most simple and well-established technologies if we hit a problem.

We hit a problem right after launch. Our WebSocket server implementation started behaving very strangely under the sudden and heavy real-world usage of launching at TechCrunch disrupt, and we were glad to be able to revert to plain polling and tune server performance by adjusting the active and idle polling intervals. It allowed us to degrade gracefully as we increased from 300 to 50,000 users in under a week. We're back on WebSockets now, but having a working short-polling system still seems like a very prudent fallback.

The Server

- node.js
- HAProxy
- Redis
- MongoDB

NODE.JS

The server side of Trello is built in Node.js. We knew we wanted instant propagation of updates, which meant that we needed to be able to hold a *lot* of open connections, so an event-driven, non-blocking server seemed like a good choice. Node also turned out to be an amazing prototyping tool for a single-page app. The prototype version of the Trello server was really just a library of functions that operated on arrays of Models in the memory of a single Node.js process, and the client simply invoked those functions through a very thin wrapper over a WebSocket. This was a very fast way for us to get started trying things out with Trello and making sure that the design was headed in the right direction. We used the prototype version to manage the development of Trello and other internal projects at Fog Creek.

By the time we had finished the prototype, we were good and comfortable in Node and excited about its capabilities and performance, so we stuck with it and made our Pinocchio proto-Trello a real boy; we gave it:

- a real DB and Schema (node-mongodb-native and Mongoose)
- basic web tech like routes and cookies (Express and Connect)
- multiple server processes with zero-downtime restarts (Cluster)
- inter-process pubsub and structured data sharing via Redis (node_redis)

Node is great, and getting better all of the time as its active developer community churns out new and useful libraries. The huge amount of continuation passing that you have to do is an issue at first, and it takes a couple of weeks to get used to it. We use a really excellent async library (and the increased code brevity of CoffeeScript) to keep our code under control. There are more sophisticated approaches that add features to JavaScript to automate continuations, but we're more comfortable with just using an async library whose behavior we understand thoroughly.

HAPROXY

We use HAProxy to load balance between our webservers. It balances TCP between the machines round robin and leaves everything else to Node.js, leaving the connections open with a reasonably long time to live to support WebSockets and re-use of a TCP connection for AJAX polling.

REDIS

Trello uses Redis for ephemeral data that needs to be shared between server processes but not persisted to disk. Things like the activity level of a session or a temporary OpenID key

are stored in Redis, and the application is built to recover gracefully if any of these (or all of them) are lost. We run with allkeys-lru enabled and about five times as much space as its actual working set needs, so Redis automatically discards data that hasn't been accessed lately, and reconstructs it when necessary.

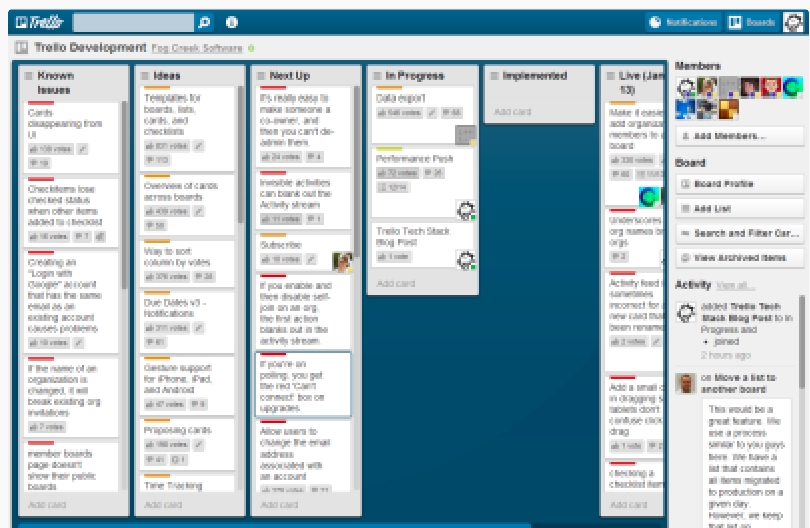
Our most interesting use of Redis is in our short-polling fallback for sending changes to Models down to browser clients. When an object is changed on the server, we send a JSON message down all of the appropriate WebSockets to notify those clients, and store the same message in a fixed-length list for the affected model, noting how many messages have been added to that list over all time. Then, when a client that is on AJAX polling pings the server to see if any changes have been made to an object since its last poll, we can get the entire server-side response down to a permissions check and a check of a single Redis value in most situations. Redis is so crazy-fast that it can handle thousands of these checks per second without making a substantial dent into a single CPU.

Redis is also our pub/sub server, and we use it to propagate object change messages from the server process making the initiating request to all of the other server processes. Once you have a Redis server in place, you start using it for all sorts of things.

MONGODB

MongoDB fills our more traditional database needs. We knew we wanted Trello to be blisteringly fast. One of the coolest and most performance-obsessed teams we know is our next-door neighbor and sister company StackExchange. Talking to their dev lead David at lunch one day, I learned that even though they use SQL Server for data storage, they actually primarily store a lot of their data in a denormalized format for performance, and normalize only when they need to.

In MongoDB, we give up relational DB features (e.g. arbitrary joins) for very fast writes, generally faster reads, and better denormalization support — we can store a card's data in a single document in the database and still have the ability to query into (and index) subfields of the document. As we've grown quickly, having a database that can take a fair amount of abuse in



terms of read and write capacity has been a very good thing. Also, MongoDB is really easy to replicate, back up, and restore (the Foursquare debacle notwithstanding).

Trello Today

Another neat side benefit of using a loose document store is how easy it is to run different versions of the Trello code against the same database without fooling around with DB schema migrations. This has a lot of benefits when we push a new version of Trello; there is seldom (if ever) a need to stop access to the app while we do a DB update or backfill. This is also really cool for development: when you're using hg (or git-) bisect and a relational test DB to search for the source of a bug, the additional step of up- or downgrading a test db (or creating a new one with the properties you need) can really slow things down.

So we like it?

We like our tech stack. As Joel observes, we've bled all over it, but I've never seen a team make an interesting app without tool- and component-related bloodshed, and not everyone can say that they really like what they've ended up with. As is true of most applications, no component or implementation detail is necessary to its nature; however, we think that this excellent set of open-source projects has sped up our development, left us with a solid and maintainable code base that we're eager to move forward with, and made Trello a more responsive and beautiful app. Thanks to everyone who has contributed to them; it's a great time to be a programmer.

Sound neat? Try Trello! It's free.

Just can't get enough tech stack talk? Here's a Prezi I made for a recent talk on Trello.

** The Socket.io server currently has some problems with scaling up to more than 10K simultaneous client connections when using multiple processes and the Redis store, and the client has some issues that can cause it to open multiple connections to the same server, or not know that its connection has been severed. There are some issues with submitting our fixes (hacks!) back to the project – in many cases they only work with WebSockets (the only Socket.io transport we use). We are working to get those changes which are fit for general consumption ready to submit back to the project.*

More on Fog Creek:

Make Better Software Magazine

From Creek Week To Final Product: A Tale of Two Projects

From Idea to Product with Creek Weeks

Announcing HyperDev - The Developer Playground for Building Full-stack Web Apps, Fast

Iteration Planner Improvements

Read the best from 16 years of Fog Creek

Business / Design / Programming / Support / Resources / Everything

---

Make Better Software Magazine
Expert advice about development, hiring and leadership

Download for Free →

Keep up with the latest

you@email.com Subscribe

Search the Blog

© 2016 Fog Creek Software, Inc.