human javascript

Henrik Joreteg

1. *We're not done*
2. *Staying up to date*
3. *Complimentary Resources*
4. *Open source is a group effort*
5. *Feedback on the book*
6. *Thank you, thank you, thank you*

Foreword

The first thing I learned to do with JavaScript was swapping images.

It was 1998 and that was the “state of the art” web application programming that I had any access to. It was a time before the web was filled with `MM_swapImage`, although I remember Dreamweaver being around. I knew HTML back then and hacking some JavaScript was my first foray into programming proper. The fact that the `src` of an `<img>` was something that I could programmatically change was a profound realisation for me at the time, although that now seems a very long time ago.

My web hacking went along with what everybody else was doing, maybe with a delay of a year or two because I had to wait for stuff to arrive in Germany (the Internet was very slow back then). Netscape and IE 4 came out and brought new ways of dynamically manipulating and animating HTML content. DHTML was born, and I tried to figure it all out until I could swap out content with layers (layers!) and divs that worked in *both* browsers.

It was cool to play with, but at that time building things that worked

consistently was not much fun, so I turned to the dark side and got involved with backend web development. I could let others worry about all the quirky browser differences. The backend is where I found my passion for databases.

But then 2005 came around and everything changed. Google unveiled Maps and the world was a different place. I distinctly remember, to this day, dragging the map inside the browser window — and I still get goosebumps thinking about it. Google Maps was the Beatles coming to America moment for the web.

Around the same time, 37signals came along with their web based apps for freelancers and small businesses and not only showed the world what can be done on the web, they started a year long discussion between supporters of what we call native desktop apps and web apps. We now go through the same thing with native mobile apps and the web.

37signals even wrote a book about how to build a bootstrapped business around the software as a service model that sparked a whole generation of web app entrepreneurs who built great products that touched the world in their own ways.

Truly inspired, a friend from university and I set out to do our own software as a service. We spent a good amount of time on it, but ultimately, it didn't go anywhere. There were a number of issues, but my poor understanding of frontend web technologies certainly didn't help. We managed to build a functional prototype, but the code was a mess, bugs were impossible to fix, new features wouldn't land, and refactoring efforts failed multiple times until we gave up. We folded and I went back to the backend and database side of things.

Fast forward to where I start Hoodie and find myself in the company of

excellent frontend developers. Now, the best I can do for the project is build out all the backend so they don't have to worry about a thing while they knock out the frontend.

However, I am still longing for a deeper understanding of the frontend world. I can dabble, but I want to be confident.

Computer science is fundamentally about building things. Unlike architecture or structural engineering, the what, how, and why of doing our work changes on a scale of years, if not weeks sometimes. Stuff we learned last year is proverbially obsolete today.

The flip side of this nature is that it is very hard to build a canon. Sure, we have algorithms and data structures as the foundations of modern teaching and yes, they are part of everyday work, but programming and building products is just *so much more*.

It is on us to figure how to get the missing pieces into a curriculum that informs future generations of web developers.

When Henrik told me he was working on Human JavaScript I begged to be added to the list of reviewers. Not because I thought I could point out all the places where he is wrong, but because I am still curious about the field of frontend web developers. I also liked the angle, thinking "I'm a human, this book is for me!" Joking aside, technical books tend to be very dry. This one promised to have a different approach, one that I could relate to.

I read the draft in one sitting, took lots of notes and at the end of it, this whole space of the frontend world that wasn't accessible to me, became clearer. I saw the necessity for more complex MVC frameworks on the client than I had experience with. I started to understand why structuring your code in certain ways makes it more robust in the face of having multiple people work on it. The book managed to illuminate well known

concepts in the unfamiliar frontend territory. Human JavaScript made frontend development accessible to me.

Truly inspired again I went on and built a small side project. It isn't much, but it makes me excited about building more web apps.

Human JavaScript is the collected wisdom of the team at &yet, where Henrik works, and all the blood and sweat that goes into their excellent client projects, and their own products. It couldn't come from a better group of people. They care about the right technology as much as teaching it. They take it upon themselves to fill the gaps of missing canon for web development.

&yet are the 37signals of our generation and I just can't wait to see what the people they inspire come up with. I can't wait to see what you come up with.

— Jan Lehnardt

Berlin, September 27th 2013

Jan is an Open Source Software developer from Berlin, Germany. He is the Vice President of Apache CouchDB, the database that replicates. He's the co-curator of JSConf EU, a partner at The Node Firm and his one famous JavaScript project is mustache.js. He currently works on hood.ie, an open source noBackend solution that aims to do for frontend development what Rails did for the backend: make all the tedious pain go away, and hide it behind an intuitive JavaScript API.

Human JavaScript

Code is as much about people as it is about computers. Sure, it's run by computers, but it's written by, maintained by, and ultimately created for people. People are not computers. We are not robots. We are unpredictable, flawed, and irrational. The same people with the same tools and instructions won't produce the same output each time. We generally don't like being alone and we don't work well in isolation. In fact, in order to do our best work we *need* to work with other people. None of these traits are bad things, quite the opposite. They're what makes us who we are, they make us, well… human. Yet, as developers it's easy for us to get so focused on optimizing for technology that we forget to optimize for people.

You can read about JavaScript, the language, elsewhere. Its good parts, bad parts, and ugly parts are well documented. This is a book about a specific set of tools, patterns, and approches that we feel are optimized for people. These approaches enable our team to quickly build and deliver high-quality JavaScript applications for humans.

&yet, the team that I'm humbled to be a part of, is a small (~20 person) bootstrapped consulting and product company focused heavily on realtime

single page web applications. We've had the opportunity to build a very broad range of single page applications for all kinds of purposes and audiences. We've built stuff for mobile, desktop, browser extensions, PhoneGap, televisions, you name it. From these experiences patterns start to emerge. Patterns that enable us to efficiently ship real-life applications (with real-life deadlines) as a team.

As we've gone along, we've done our best to extract reusable tools out of them. So, in some ways we accidentally wrote this book. What I mean is that much of its contents are compiled from past blogposts, explanations to teammates and clients, and from project README files. This book is primarily an extraction, not a creation. We're sharing our experience, secrets, and tools to hopefully give you and your team a solid footing for building great apps and experiences.

Acknowledgements

Speaking of humans, this book would not exist if not for a giant list of people who helped make it a reality. To name a few specifically a huge thank you to Jan Lehnardt, Philip Roberts, Bear (Mike Taylor), Luke Karrys, Jenn Turner, Adam Brault, Sara Chipps, Jeff Boyus, Lance Stout, Karolina Szczur, Jon Hjelle, Melanie Brown, Stephanie Maier, Beau Sorensen, Michael Garvin, Amy Lynn Augspurger and others.

They helped with technical review, code, editorial feedback, design, writing/producing the promotional video and loads of encouragement. Thank you all!

This Book Will Help You Build Native HTML5 Apps

Let's talk about this whole "app" thing for a bit and get on the same page in terms of terminology. If you own a smart phone of any sort, you've been inundated with the word "app" over the past five years or so.

Rather than pontificate on the meaning of this for three chapters, I'll explain the distinction as I see it for the purposes of this book.

When most people say they're building a "web app" they're talking about writing source code that describes an application that will run on the server and send rendered HTML to the browser. That definition seems a bit narrow and limiting. Plus, when I think of my favorite web apps they don't fit neatly into that box. The best web apps often have multiple interfaces and clients, some native, some web. Most play nicely with or completely integrate with other services. Generally web apps are good at solving some specific problem or provide some specific benefit and use the web to tie it all together.

The web vs. native debate is a bit worn out. From my perspective the whole debate is somewhat misguided. It doesn't have to be one or the other, I have no problem with it being both. It's no secret that most native apps are even better with the web. Why else would classic native Apple apps like iPhoto start integrating with Flickr and Facebook? And yet, you can't write a native app for every platform out there.

Thinking of your "web apps" as an API with a series of clients seems much more fitting. It just makes sense. Your API defines your service, connects you to other users, and ties in the whole experience. Then you can focus on building clients that provide the best experience possible for various environments and uses.

Let's talk about browsers for a second. They're completely freakin' amazing. Well, the modern ones, at least. They are nothing short of extremely capable, *mostly* standardized operating systems that are freely available on nearly every platform. They keep getting more and more amazing every day. Sadly, the addiction to backwards compatibility is crippling perceptions of what that operating system is capable of. Too often the web interface ends up being the lowest common denominator in terms of experience. That doesn't have to be the case! Let's build for the future of the web, not its past.

The types of apps we're talking about building in this book could really be called "Native HTML5 apps" in that they use HTML5 to its full extent without bowing to compatibility with crappy old browsers.

To clarify further:

1. They are separate from the API.
2. They don't work *at all* if someone has JavaScript turned off.
3. A modern browser with a modern JavaScript engine is a minimum

system requirement.
4. We send the application code itself to the browser, not the result of running the application code.
5. The app is rendered entirely on the client. We only send the bare minimum HTML we need to tell the browser to run our app. Usually just a doctype, a script tag and a stylesheet.
6. After loading, the client fetches its own data, as data (typically JSON), not as rendered HTML.
7. The app is loaded once and never does a full page reload while you're using it.
8. Actions like clicking on a link to render a new "page" utilizes asynchronous JavaScript.
9. The app has and maintains "state" that is cached and maintained separate from the server.

From now on, when I say "app" or "native HTML5 app" or "browser app" or "client app" within these pages, that is what I'm referring to.

Here's more food for thought: once you acknowledge that the browser has state, you really ought to think about how to keep that state up to date and make it a "realtime" application.

Realtime apps are human apps

A lot of people get hung up on the term "realtime." The way I'm using it here is not referring to latency or speed of delivery, it's about the fact that there are multiple sources of data (usually people) doing stuff! People are changing the data all the time and our app isn't passively waiting for the user to refresh. Instead the app keeps itself up to date. In reality, no web app is "hard real-time" as per computer science. In our case, it's a term to

help describe apps that keep themselves up to date.

Realtime isn't about hype and technology – it's about removing the friction of technology on collaboration and overcoming the confusion of keeping track of lots of state.

The future of the web is realtime. Of this I have no doubt.

The reason I can say this with such certainty is that it's already happening under our noses.

Facebook, Gmail, Google Talk, and GitHub just to name a few, have all implemented some form of automatic page updating. When they have something new to tell you, they don't wait for you to ask for it. They push it out to you, from the server to the client.

In some cases this is as simple as the page automatically polling to see if there's something new. In other cases it's more advanced, where all the data used to build and update the page is coming over an open WebSocket connection. For our purposes, the transport mechanism is largely irrelevant; the point is, data comes to you.

This inherently breaks the statelessness of the web. It used to be that I hit a URL and got back a webpage. As a user I understood that the information on the page was (probably) accurate as of the time it was requested. If I wanted to check for something new, I'd go ask for it again and receive another snapshot in time.

As soon as we make any effort to keep the information on the page in sync with the server, we've now acknowledged that the webpage has "state." In some ways, the page always *had* state, but it was clear to users that it was snapshotted state, not up-to-date-synchronized state. As a result, that static page was more like a printed page than a living document.

One of the fundamental advantages that digital media has over print is that it's no longer static. It's dynamic, it's fluid, and it can be updated as the information changes.

So, as soon as we as developers decide that we want to do partial updates of the page, the only way we can do so is by knowing what information we currently have, and comparing it to what's on the server. State duplication has occurred and we're now maintaining "state" in some form in the client.

As users get increasingly comfortable with that idea, I believe we'll reach a point where always-current, self-updating information is the *expectation* rather than a surprise. Facebook with its chat, live comments, and push notifications is already conditioning an entire generation of users to expect realtime updates. I believe that knowing how to build realtime apps is a crucial skill for web developers who want to stay at the top of their field.

Anytime you duplicate state, you increase complexity. Rather than worrying about just rendering some data correctly, you're now caring about staleness, caching, and conflicts.

If we step back a bit we start to realize that what we're actually building is a distributed system and as a result we'll face all the same challenges that come with building distributed systems.

I know what you're probably thinking. Some framework is going to come along that solves this problem for me. You may be right, there are many different approaches to dealing with the problems of duplicated state. There are several emerging frameworks, such as Meteor and Derby, that aim to simplify the process of building apps that work this way.

The challenge with some of those frameworks, from where I sit, is that there's a lot of emphasis on trying to share code and logic between the client and the server. In my opinion, client and server really should be

performing fundamentally different roles. Servers are for data, clients are for presentation. To me, this is about the basic principle of separation of concerns. A contrast to this is what my friend Owen Barnes was working on with SocketStream. It's funny to see how we both ended up reaching very similar conclusions over the last few years. As he mentioned in his talk at RealtimeConf 2012, there likely isn't going to be a "Rails of realtime." The problems are simply too diverse. He's since moved his focus elsewhere, but the conclusion seems to be building loosely coupled modular approaches and patterns that can be substituted as needed.

Distributed systems, latency compensation, and state duplication are really complex problems. The way you solve complex problems is by *not* solving the complex problems. Instead, you break them down into smaller, simpler, solvable problems. Those solutions in aggregate can represent the complete solution.

So, why bring the complexity of the server to the client and vice/versa? In addition, when you try to share too much server code with a browser it's very easy to fall into the trap of tightly coupling your application to that particular client. This makes it much harder to build other clients, say, for example, an iOS app for your app. While these frameworks are useful for standard desktop web apps, they let us down a bit when we want to go beyond that. With more and more talk of "the Internet of things" we have good reason to believe that the breadth of device types that want to talk to your app will continue to increase.

Misconceptions, FUD and engineering

We need to stop dumbing down the concept of "frontend" code. It's getting better, but many self-described "real" developers still think browser code is

sissy stuff. In their minds, the client is easy and it's what the designer-y, non-developer folks do. That's ridiculous.

We're not talking about rendering some HTML on the server and sprinkling on a few lines of jQuery. We're talking about *engineering* a UI here.

Unfortunately because of those pre-conceptions, many of the people who are being asked to build these kinds of apps don't have a heavy engineering background and approach the task much like they would any other client code: write some jQuery.

But, jQuery is not an application framework. It's an abstraction layer and toolkit for working with the DOM. I'm not dogging on jQuery at all, in fact, I think it's a great toolkit for DOM manipuation. I use it and am quite happy with it in many HTML5 apps. My point is simply that jQuery is a DOM toolkit, not an application framework.

But, inevitably a "frontend" person is asked to build an immersive, complex client app and soon they end up with a 3500 line JavaScript file called "app.js" that does *everything.* Also, now no one else knows how any of it works, or how it's structured. No one wins.

To avoid those situations we have to approach it as an engineering task of building a performant, well-structured UI.

Picking your tools

There are more and more tools out there now to build client apps. AngularJS, Ember, Backbone, Sencha, Knockout, etc.. all have their pros and cons.

People in charge of development teams seem to agonize over the decision.

They see these options as long term decisions with huge, long term ramifications and they don't want to pick the wrong one. The awesome thing is this…they're all JavaScript. So it's not really *that* grave of a decision and switching to something else isn't going to burn your whole business to the ground. The most important thing is that your team becomes familiar with building well-structured apps in JavaScript. That investment will be well worth it and will translate to new tools, if they come along.

Decisions are time consuming and expensive. At &yet we've built and re-built applications with all kinds of different tools and approaches. The following pages contain the conclusions we've reached. They're probably not for everyone but we've been quite happy with the results and it has made it possible for us to efficiently collaborate on clientside apps as a team. The approaches were picked with the following criteria:

1. Tools that are "just JavaScript." Not tools where you describe your app in a DSL (no Sencha). This is to avoid requiring too much knowledge of the framework itself before being able to contribute. Focusing on JavaScript also offers some protection against investing too heavily in framework-specific knowledge.
2. Tools where you build the app by writing code in JavaScript files, not by declaring bindings in your HTML (no AngularJS, sorry). Having to write application logic inside of a template feels like a violation of separation of concerns. It has some short-term payoffs and can make simple things really easy. However, when you want more control it can be difficult to do within the constraints of the framework.
3. No monolithic, do-everything widget frameworks (no Sproutcore). These often make lots of assumptions about how you want to structure your HTML and often violate separation of concerns.

4. Model state is completely decoupled from view state (no Knockout.js). Again, this is to separate concerns.
5. You should not have to be a JavaScript rockstar to edit templates. Templates in separate files with very little logic allows designers to edit templates without having to know how everything works.
6. The DOM is simply a view of the state and reacts to changes in the model layer.
7. Simple, decoupled file structures with lots of components that solve one problem.
8. As little magic as possible (no Ember). Similar to Item 1 this is primarily to avoid requiring too much framework-specific knowledge. Which brings us to the next point.
9. People who already know JavaScript should be able to work on the app without lots of knowledge about a specific tool or framework.
10. The inverse of the previous point should also be true in that people who learn how the app works, should accidentally learn how JavaScript works in the process.
11. It should play nicely in a team environment using version control (no giant files).
12. Every piece of functionality should have an obvious “home.” Structure, structure, structure. This makes it easy to jump into old code to fix bugs or to jump from project to project.
13. The project should have a set of code style standards that are enforceable by an automated process. This encourages readability and consistency throughout the codebase. It centralizes code style arguments around an enforceable standard. We find that this minimizes a lot of back-and-forth about code style because it becomes a simple automated pass or fail.

Now let’s dive in.

Client or server? Go big or go home.

As someone who writes lots of JavaScript, you might think I'd advocate that everything should be a single-page app. In short: No.

Make things as simple as you possibly can. Programming is complex, expensive, and time consuming. Pragmatism is the only way to finish anything.

For many types of applications, building a single-page app is harder and gives you no additional value. I do think there will be a day when that's no longer the case. But we're not there yet.

So, instead, think about how you want your app to be used. Is it something that a user is going to load once and leave sitting on their desktop all day? Or is it something that's quickly referenced and then closed?

As engineers we over-engineer things ALL THE TIME. Just think how many blogs hit a database with each and every request when really they could just be static HTML, generated from markdown or something (example: Jekyll).

Building client-side apps is often more complicated than a server-side rendered app. Decide carefully. Ask yourself, is there additional benefit for your users? Are you building something that is opened and closed frequently, or are you building an experience? How often does the data in the application change? Do you care if it changes while the app is open?

So I didn't talk you out of it? Ok, then go all out!

`<patronizing tone>` So you've heard of "separation of concerns" `</patronizing tone>`. We're taught to build tools and components that have a simple job and are self-contained. It makes code more reusable and more maintainable, and keeps developers more sane.

HOWEVER, the first thing people most commonly do when building web apps is render a bunch of HTML on the server, then send it to the client and start shuffling it around with a bunch of JavaScript! Just like that we've coupled server and client code because both of them now have to care about rendering and presentation.

Pick one or the other, seriously. If you're building an "app" where a significant portion of the data will be rendered on the client, just freakin' render *all* of it on the client. Don't mess around. It's just ugly to have to send a bunch of partially rendered HTML to the client and then start mucking around with it.

One of my favorite things to show developers from the And Bang codebase is the HTML we send to the browser. Here it is… in its entirety as of this writing:

```
<!DOCTYPE html>
<!-- served with <3, &yet -->
```

```
<script>window.times = {start: new Date};</script>
<link href="/&!.css" rel="stylesheet">
<script src="/&!.js"></script>
```

Yup. (And yes, omitting `<html>`, `<head>`, and `<body>` is allowed by the HTML specs.)

Am I crazy? Probably.

But, if we've decided that we want our server to be able to focus on data we might as well transfer as much of the rendering and presentation of client, to the client.

Ok, I'll admit it, some of this extreme minimalism is due to the aesthetic of it. But, it also makes it abundantly clear that it's the client's responsibility to render the application and manage everything within it, including things like the page title and life-cycle of all the document elements.

Some devs advocate partial rendering on the server for faster load times, etc. To me, once we recognize we're building a "thick" client, we might as well render it all there. Otherwise, if part of our page state is rendered on the server, somehow we have to re-translate the HTML into state information. Again, I prefer an explicit separation of concerns.

I can hear the screams now, "What about load times and performance?!?!" I'd rather optimize how my application runs once it's loaded, than shave milliseconds off the time required to download the initial app. Also, let's keep in mind that the shiny, retina-ready logo or the the background texture images you used is likely to take up as much bandwidth as your entire application. Anyway, if your app is contained in a single file (more on that later) it's fairly simple to minify, version, and tell the browser to cache it permanently so your app only downloads it once per revision.

Also, if you're pragmatic about this, then you'll recognize that at some point

you’re probably going to render some kind of signup or login form. If you’re sneaky, you could use the inevitable time the user is going to spend on that page the first time to download the app so that when users are logged in they’ll already hit it with a primed cache, even on the first load!

I’ll cover some more tools and approaches for this in Chapter 4 when discussing code organization.

Writing code for humans.

- Code is read more often than it's written.
- If you're too clever, you'll forever own the project because no one else will know what the heck you're doing. That will suck, and so will your project.
- As the requirements change and evolve (as they most certainly will), your ability to quickly read and understand the various pieces of your app will dramatically affect how quickly you can change course.

All of this is to say: WRITE CODE THAT IS EASY TO READ!

Tools and trickery

You with me? Ok, but how do you actually do this?

Well, let me give you a silly example:

```
// Assume this is an array of strings from somewhere
var myArray = ['hello', 'something', 'awesome'];
```

```
if (~myArray.indexOf('hello')) {
  // Under what circumstances does this get called?
}
```

Can you explain to me, in plain English, what that tilde does? If you can, good for you, but do you think your whole team can?

Now, compare it to this:

```
// Same array:
var myArray = ['hello', 'something', 'awesome'];
if (myArray.indexOf('hello') == -1) {
  // Pretty freakin' clear, AMIRITE!?
}
```

Or even this, using Underscore:

```
// Same array:
var myArray = ['hello', 'something', 'awesome'];
if (_(myArray).contains('hello')) {
  // Also pretty freakin' clear right?
}
```

Frankly, I think the first example looks better, visually. In fact, I sometimes use the first when working on a library that isn't meant to be a team project. However, if I'm working on an app that other people will be working on with me, I will write it the second way, because it's more explicit and requires less of the other developers who may not be familiar with the syntax in the first example.

Cleverness is a double edged sword

Being clever is sometimes a good thing. But as was so aptly put by Paddy Foran cleverness for the sake of cleverness should be avoided at all costs.

The goal should always be clarity and readability.

Code Linting

As with readability, code conventions and format should be consistent throughout the project. In practice, if you have multiple people involved in a project, this can be hard.

Semicolons, tabs, and spaces are contentious things among developers. Every developer I've ever met has opinions (usually strongly held) about code style.

If you're building large JS apps and not doing some form of static analysis on your code, you're asking for trouble. It helps catch silly errors and forces code style consistency. Ideally, no one should be able to tell who wrote what part of your app. If you're on a team, it should all be uniform within a project. How do you do that? We use a slick tool written by Nathan LaFreniere on our team called, simply, precommit-hook. So all we have to do is add "precommit-hook" to our list of dependencies (in a Node project).

What it does is install a git pre-commit hook in the project that uses JSHint to check your project for code style consistency before each commit. Once upon a time there was a tool called JSLint written by Douglas Crockford. Nowadays there's a less strict, more configurable version of the same project called JSHint.

The neat thing about the npm version of JSHint is that if you run it from the command line it will look for a configuration file (.jshintrc) and an ignore file (.jshintignore), both of which the precommit hook will create for you if they don't exist. You can use these files to configure JSHint to follow the code style rules that you've defined for the project. This means that you can now run `jshint .` at the root of your project and lint the entire thing to make sure it follows the code styles you've defined in the `.jshintrc` file. Awesome,

right!?!

Our `.jshintrc` files usually looks something like this:

```
{
  "asi": false,
  "expr": true,
  "loopfunc": true,
  "curly": false,
  "evil": true,
  "white": true,
  "undef": true,
  "indent": 4
}
```

The awesome thing about this approach is that you can enforce consistency, the rules for the project are contained, and actually get checked into the project repo itself (in the form of the `.jshintrc` file). So, if you decide to have a different set of rules for the next project, fine. It's not a global setting; it's defined and adjusted by whomever runs the project. Optionally, you can also specify your jshint config in package.json by adding a `jshintConfig` property containing the same type of config as above.

For a more in-depth discussion on style and style guides I highly recommend reading Airbnb's JavaScript style guide. It will give you a good overview of the various common style discrepancies and the reasoning behind some of their choices. It's also a great starting point if you want to fork it and tweak it to be "the style guide" for your team.

No more clientside spaghetti. Organizing your code.

The single biggest challenge you'll have when building complex clientside applications is keeping your codebase from becoming a garbled pile of mess.

If it's a long-running project that you plan on maintaining and changing over time, it's even harder. Features come and go. You'll experiment with something, only to find it's not the right call and leave traces of old code sprinked throughout.

I absolutely *despise* messy code. It's hard to read, hard to maintain, hard to collaborate on, and it's just plain ugly to look at. Beyond those pragmatic reasons, I consider my code to be my craft. Therefore, I want the care that I put into writing it to be obvious to those who read it.

Complexity sneaks up on you. If you don't actively fight for simplicity in software, complexity will win.

Here are a few techniques, crutches, coping mechanisms, and semi-pro

tips for staying sane.

Refactor early, refactor often

Entropy is inevitable in a codebase. If we don't continually modify, simplify and unify the existing code along with the new code that's being written, we can easily end up with a really big, messy app.

Some developers seem hesitant to touch code they've already written. But, I believe that deleting and updating code is a regular and important part of building an app. When you first start building an app, you don't know how you're going to build everything in it so there's no reason to treat any of the code you build along the way as infallible.

Code is just text, not an edict. It can be changed easily and should be streamlined as you build.

Don't be scared of refactoring. Be scared of building an unmaintainable piece of crap. I have found that to be much more costly in the long run. Additionally, if your app is separated into clean simple modules the risk of accidentally breaking something else is dramatically lower.

Separating views and state

This is the biggest lesson I've learned building lots of single page apps. Your view (the DOM) should just be blind slave to the model state of your application. For this you could use any number of tools and frameworks. I'd recommend starting with Backbone (by the awesome Mr. @jashkenas) as it's the easiest to understand, and the closest thing to "just JavaScript"™

as discussed in the introduction.

Essentially, you'll populate a set of models and collections of these models in memory in the browser. These will store all the application state for your app. These models should be completely oblivious to how they're used; they merely store state and broadcast their changes. Then you will have views that listen for changes in the models and update the DOM. This core principle of separating your views and your application state is vital when building large apps.

One aspect of this approach that is commonly overlooked is the flexibility it provides if you decide the app should have a different UI (`<sarcasm>`which *never* happens, right?!`</sarcasm>`), or if you build another application on the same API. All of the models pretty much without modification are completely reusable.

CommonJS Modules

I'm not going to get into a debate about module styles and script loaders. But I can tell you this: I haven't seen any cleaner, simpler mechanism for splitting your code into nice isolated chunks than CommonJS modules.

Let's pause for just a second to discuss what modules do for us. JavaScript has globals. What I mean is that if you don't put a `var` in front of any variable declaration, you've just created a global variable that's accessible from *any* other code in your app. While this *can* be used for good it also gives you a lot of rope to hang yourself with. Without a way of managing this, as your app grows, knowing what global variables you have at what time will become nearly impossible and will likely be a big source of bugs. We also want to build our app in tiny pieces of independent code

(a.k.a. modules). So, how do we make sure each module has access to what it needs? By not referencing globals and by having each module explicitly `require` other code that it needs. That's why we need a module system. Very few things will have a greater positive impact on your code structure than switching to a good module system.

CommonJS is the same style/concept that is used in Node. By following this style you get the additional benefit of being able to reuse modules written for the client on the server and vice versa (though, the overlap is usually not that big).

If you're unfamiliar with the CommonJS modules style, your files end up looking something like this:

```
// You import things by using the special `require` function and you can
// assign the result to a variable
var HumanModel = require('human-model');
var _ = require('underscore');

// You expose functionality to other modules by declaring your main export
// like this.
module.exports = HumanModel.define({
  type: 'navItem',
  props: {
    active: ['boolean', true, false],
    url: ['string', true, ''],
    position: ['number', true, 200]
  },
  init: function () {
    // Do something
  }
});
```

That's it! Super easy. You don't create any globals. Each file that uses your module can name it whatever makes the most sense for use in that module.

You just export a constructor (like above), or a single function, or even a set of functions. Generally, however, I'd encourage you to export only one thing from each module.

Of course, browsers don't have support for these kinds of modules out of the box (there is no `window.require`). But, luckily that can be fixed. We use a clever little tool called browserify that lets you `require` whatever modules you need. This also includes being able to declare dependencies in a `package.json` file and just installing `require()`-able modules from npm into your project. With this approach, no clientside specific package management like Bower is required. You simply declare your dependencies in your package file and install them.

Browserify will create a `require` system and starting with the module you specify as an entry point it will include each `require`-ed piece of code into an app package that can be sent to the browser.

Browserify is written for Node but even if you're using something else to build your web app, you can use Node and browserify to build your client package. Ultimately, you're just creating a single JS file. So once it's generated, that file can be served just like any other static file by any file server you want.

Grab your moonboots

If you're used to building apps where each script in your app directory has a corresponding `<script>` tag hardcoded in some HTML file somewhere it can be a bit confusing when switching to using a script module system like browserify.

As I touched on in Chapter 2, we really would like our production environment to serve a single, minified, `.js` file with a unique file name so that we can tell browsers to cache it forever. However, that's far from ideal in a development environment because we don't want to debug minified

code in the browser or have to rebuild it with every change. So, in the interest of keeping the development cycle enjoyable here's what we want:

1. Easy way to edit/refresh your clientside JavaScript files without having to restart the server or re-compile anything manually.
2. Be able to easily map code in your browser to the right file and line number in the non-compiled version in your app folder.
3. Serve unminified code in development.
4. In production, serve a minfied, uniquely named, permanently cachable file containing your entire app.
5. Be able to toggle between those two states with a simple config flag.
6. Be able to use browserify for all compatible modules, but still be able to bundle other libraries into our app file.
7. Be able to serve/minify/cache CSS in the same way.

Since defining this type of browser app "package" is such a common problem that we want for all apps, I built a helper to make it a bit easier to work with.

It's called "moonboots." To use it, you define your browser app like this (assuming Node and Express):

```
var Moonboots = require('moonboots');


var clientapp = new Moonboots({
  // The directory where all the client code is stored
  main: __dirname + '/clientapp/main.js',

  // Whether or not to build and serve cached/minified version of
  // the application file.
  // While you're in development mode you don't need to restart the
  // server or do anything other than edit clientside code in your project.
  developmentMode: true,

  // These are the regular JavaScript files (not written in CommonJS style)
  // that we want to include in our application. These all live in clientapp/libraries
  // and will be concatenated in the order listed.
  libraries: [
    __dirname + '/libs/jquery-1.9.1.js',
    __dirname + '/libs/jquery.plugin.js'
  ],
```

```
  // These are our stylesheets. They will be concatenated and run through
  // cssmin to minify them.
  stylesheets: [
    __dirname + '/public/css/styles.css'
  ],

  // We pass in the Express app here so that it can handle serving files during development
  server: app
});
```

At this point we can tell Express the routes where we want it to serve our application. This is a bit hard to wrap your head around if you're not used to single page applications that do clientside routing.

Since we're sending a JavaScript application, rather than rendered HTML to the browser, it's going to be up to the client to read the URL, grab the appropriate data, and render the appropriate page represented by that URL. So it's up to us to configure our server to always respond with the same HTML at any URL that is considered part of our client application. We cover the concept of clientside routing in a bit more detail in Chapter 9.

You can do this in Express through the use of wildcard handlers, or by passing regular expressions instead of strings as the route definition. If you look at the sample application you'll see the relevant line in server.js looks like this:

```
app.get('*', csrf, clientapp.html());
```

Where `clientapp` is the app we defined above. Calling `html()` on it will return a request handler that serves up the base HTML for the application at all the relevant routes. By simply having the helper provide a request handler, you can still add whatever middleware you want first (as seen with CSRF in that example).

The need for the wildcard URL becomes more obvious in your application when you open it and navigate to a different URL within an app that uses

HTML5 push state. Say we click a button that takes us to `/sample` within the app. When navigating to that page, the browser won't make any server requests, but you'll see the URL change. However, now that you're viewing the `/sample` page, if you refresh the browser, the browser will make a request to `/sample`. So if your server app isn't set up to serve the same response at that URL, it won't work.

A note on going to production

Node happens to be pretty good at serving static files. So just serving the production file with Node/moonboots is probably sufficient for most apps with moderate traffic. In production mode, moonboots will build and serve the app file from memory with aggressive cache headers.

However, a lot of people like to serve static files with a separate process, using nginx or using a CDN, etc. In that scenario, you can use moonboots during development and then generate the minified file, write it to disk, or put it on something like an S3 as part of your deploy process.

Calling `moonboots.sourceCode(function (source) { ... })` will call your callback with the generated source code based on current config, which you could use to write it to disk or put it on a CDN as part of a grunt task or whatnot. Those details are probably beyond the scope of this book. But, the point is, you can certainly do that with these tools if that makes more sense for your app.

The structure of the clientapp folder

Our clientapp folder usually contains the following folders:

- models (folder): Contains definitions for all backbone models and collections. As a sanity check, none of these files should have anything related to DOM elements or DOM manipulation.
- pages (folder): The pages folder is where we store the specialized Backbone views that represent a page rendered at a specific URL.
- views (folder): The views folder contains all of our Backbone views (that are not pages), so things like the main application view and views for rendering specific types of models, etc.
- app.js (file): This is the main entry point for our application. It creates an `app` global variable and instantiates the main models and views.
- router.js (file): This is our clientside (Backbone) router. It contains a list of URL routes at the top and corresponding handlers, whose job it is to instantiate the right views with the right models and call `app.renderPage` with those values.
- libraries (folder): This contains all the libraries we're using that are *not* structured like CommonJS modules. So things like jQuery and jQuery plugins will go here.
- modules (folder): Here is where we put all the clientside modules that we want to be able to require without a relative path. This is a good place to put our compiled template file:
    - templates.js (file): This is the module that gets created from the templates folder (see next). It's a single file with a function for each clientside template. This file gets auto-generated so don't try to edit it directly. Putting it in here lets us also require and use our

template functions easily within our views. Each template has a corresponding template function. Each function takes your context object and returns just a string of HTML.

- templates (folder): Here is where we keep all our Jade files that get used in the client application. Anytime you're wanting to create HTML within the app, use a Jade template and put it in here. You can structure this folder in whatever fashion makes sense for your application. The important thing to understand is that folders become part of the template.js module structure. For example, in this template you'll see that there's a `pages` folder within the templates folder with a file called `home.jade`. To use the function that got created from that, you'd access it as follows:

```
var templates = require('templates');

// Note that 'pages' becomes part of the structure of your
// imported templates object
var html = templates.pages.home();
```

See Chapter 8 for a more in-depth discussion of templating.

Creating an app global

So what makes a module? Ideally, I'd suggest each module being in its own file and only exporting one piece of functionality. Only having a single export helps you keep clear what purpose the module has and keeps it focused on just that task. The goal is having lots of modules that do one thing really well so that your app combines modules into a coherent story.

When I'm building an app, I intentionally have one main controller object of sorts. It's attached to the window as `app` just for convenience. For modules

that I've written specifically for this app (stuff that's in the clientapp folder) I allow myself the use of that one global to perform app-level actions like navigating, etc.

The main app object doesn't really need to be all that special. Often I create an object literal with a main init function (more on that in Chapter 10). But generally it will look like this:

```
module.exports = {
  // Main init function
  blastoff: function () {
    // Attach our app object to the window
    window.app = this;
    // This is where we render our main view, get some data,
    // kick off the history tracking, etc.
    // See Chapter 10 for more detail.
    ...
  },

  // Render a page view passed by the router
  renderPage: function () { ... }

  // Alias to Backbone.history object so we can
  // do app.navigate('/someother/page') from
  // anywhere in the app.
  navigate: function (url) {
    app.history.navigate(url, true);
  }
};

// Run our whole app, it all starts here:
module.exports.blastoff();
```

Note that very last line that actually calls the `blastoff()` function. That's how we kick off the whole thing. That's our main entry point to the app.

Using events: Modules talking to modules

How do you keep your modules cleanly separated? Sometimes modules are dependent on other modules but we still want to be able to keep them loosely coupled? One good technique is triggering lots of events that can be used as hooks by other code. Many of the core components in Node are extensions of the EventEmitter class. This means you can register handlers that get called when events happen to that object, much like you would do in the browser when you want to register a click handler for an element on the page.

I find that developers often assume that events are kind of magical or special things in JavaScript, but they're not. In fact, building an event emitter from scratch is a really great learning exercise. They're really quite simple. You're just saying: "Please call this function when this thing happens." Typically, you'll see code like this:

In browsers:

```
document.getElementById('something').addEventListener('click', function () { ... }, false);
```

In jQuery it looks like this:

```
$('#something').click(function (event) { ... });
// or
$('#something').on('click', function (event) { ... });
```

In EventEmitter it looks like this:

```
myEventEmitter.on('someEvent', function () { ... });
```

But they all do the same thing: they store a reference to the function you handed it (usually by adding it to an array of functions internally). Then, when the event happens, they call all the functions in the relevant array with information about the event. That's it! No magic.

This pattern is really useful when building reusable components yourself. Exporting objects and classes that inherit from some type of event emitter means that the code using your module can specify what they care about, rather than the module having to know.

At points of interest within your module where you think some external source may care, you can just call `this.emit('someEventName', {some: 'data'})` and if there are any handlers for that event, they'll be called.

There are lots of implementations of event emitters with various features. Features usually involve various ways of registering and unregistering event listeners. For example, you may want to register a handler that only gets called the first time an event happens. So for this many event handlers have a `once()` method alongside the `on()` method. In addition, some event handlers give you a way to listen to all events emitted by a certain object, or perhaps all events in a certain namespace. These features can be useful for logging out all events (so you can debug), or for proxying events from one event source to another object.

Browsers don't expose a base EventEmitter class we can just use, so for clientside code we need to include one in order to take advantage of this pattern.

We use a slightly modified version of a really awesome and lightweight one that was written by the LearnBoost guys: @tjholowaychuk, @rauchg and company. It's wildemitter on my GitHub if you're curious.

Beyond standard `on()`, `off()` and `once()` methods it adds two main features:

1. Wildcard event handlers for listening to all events in an object. For example: `emitter.on('*', function (eventName, event) { ... })` or `emitter.on('namespace*', function (eventName, event) { ... })`.

2. Grouped event handlers, meaning you can specify which group the handlers are a part of when you register them and then unregister all the handlers in the group at once:

```
var WildEmitter = require('wildemitter');

var emitter = new WildEmitter();

// Register one handler
emitter.on('something', 'group1', function () { ... });
// Register another handler in the same group
emitter.on('someOtherEvent', 'group1', function () { ... });

// Then release both of them
emitter.releaseGroup('group1');
```

Details and implementations aside the same basic concepts of adding and removing handlers are available in all event emitters.

As an example, here's a simplified version of the `andbang.js` library which is an SDK for talking to the And Bang API.

```
// Require our emitter
var Emitter = require('wildemitter');
```

```
// Our main constructor function
var AndBang = function (config) {
  // extend with emitter
  Emitter.call(this);
};

// Inherit from emitter, but retain constructor
AndBang.prototype = Object.create(Emitter.prototype, {
  constructor: {
    value: AndBang
  }
});

 // Other methods
AndBang.prototype.setName = function (newName) {
  this.name = newName;
  // We can trigger arbitrary events
  // these are just hooks that other
  // code could chose to listen to.
  this.emit('nameChanged', newName);
};

// Export it to the world
module.exports = AndBang;
```

Then, other code that wants to use this module can listen for events like so:

```
var AndBang = require('andbang');
var api = new AndBang();

// Now this handler will get called any time the event gets triggered
api.on('nameChanged',  function (newName) { /* do something cool */ });
```

This pattern makes it easy to expose functionality without needing overly specific knowledge about how it’s going to be used.

Models

A simple example

Let's say you have a list of items. When a user clicks on an item, you want to visually mark it as selected. Someone used to building simple apps would probably do something like this:

```
// Register a click handler on the parent list
$('ul.theList').delegate('click', 'li', function () {
  // Toggle a class on the clicked item
  $(this).toggleClass('selected');
})
```

So now, clicking on an item will toggle a class. jQuery's `toggleClass()` method will check whether it's already got the class, and add or remove it as necessary. Great! We're done!

Err… well typically if you're going to select something it's for a reason, right? So our app is going to want to *do* something with the selected item or items.

Let’s say the user has selected several things and now wants to delete them by clicking a delete button. No problem, you say, we just add a button handler that find the ones with the selected class and deletes them.

```
$('button.delete').click(function () {
  // Get our seletedDOM items, loop through them
  $('ul.theList li.selected').each(function () {
    // But we also have to have a way to figure out what
    // ID each one of these things represent so we can pass
    // the correct info to the server. So, let's assume we use
    // HTML5 data attributes. Luckily jQuery's data() method
    // reads all those and returns them as an object.
    var id = $(this).data('serverId');
    var listId = $(this).data('listId');
    $.ajax({
      type: 'delete',
      url: '/lists/' + listId + '/widgets/' + id,
      success: function () {
        // Do something
      },
      error: function () {
        // Let the user know, somehow
      }
    })
  });
});
```

Ok, not too terrible, you say?

Well now what if we’ve got these additional requirements?

1. There isn’t just one list, there are several on the page at once. There are some actions that can be performed in bulk, but only for some of the items in some of the lists.
2. There are some items you can’t delete, because you don’t have permission to, but you can still select them and annotate them.
3. Deleting the item is only one of six different possible actions you can take with each item.
4. You have to support full keyboard control, as well as handling mouse clicks.
5. You now want to support selecting the top item, holding shift and clicking the bottom one to select a range.

If you continue this same approach, you're in deep doo doo

You can handle adding features to a point. But you will reach a point where you start arguing against adding features, not because you don't think they're good ideas, but because you're scared to implement them because of the headaches and bugs it will inevitably cause.

Welcome to nearly everyone's first single page app experience.

Enter Models

If you've never used models in clientside code, it's not as intimidating as it may sound. The idea is simply that we create some data structures in the browser, separate from the DOM, that hold the data that we got from the server as well as any client specific data or state.

The "selected" state as described above is a good example of client-specific state, meaning when we go to update an entry in the API, we're not going to send `{selected: true}` as one of its properties. The server doesn't care about that, it's just used to track the state of the user interface.

So, what is a model anyway? What does it give us?

The fundamental thing a model should provide is observability. What do I mean? Well, in the same way you can register an event listener in the browser that responds to a form input value changing:

```
document.getElementById('myInput').addEventListener('change', function () {
  // Do something with the value
});
```

A model should let us listen for changes to its properties:

```
model.on('change:selected', function (newValue) {
  // Do something with the new value
});
```

In addition, models should contain functionality that makes it easy for us to work with that data. That means things like exposing some processed form of the data, should be a method on the model. Let's say one of the model properties represents a date. We may have a method on the model for getting a nicely formatted date string built from that date object. Arguably this is a presentation issue, but the model ends up being a logical place to expose a string version of the date property for maximum re-use and consistency.

In addition, models are a good place for methods that perform actions on the model itself, like updating the server when the model changes.

In And Bang, we do a lot with tasks. You can assign them to each other, "ship" them, "later" them, "trash" them, etc.

So, each of these actions are represented by a method on the task model that sends the correct data to the server, as well as updating the appropriate properties on the local model.

For example, here's the `trash` method of a task in And Bang:

```
...
},
trash: function () {
  if (this.deleteable) {
    this.removing = true;
    this.api('deleteTask');
  }
  return this.deleteable;
},
...
```

It includes an upfront check to see whether we even have permission to trash this item. In case you're wondering this isn't actually used to enforce this permission, that's the API's job.

In fact, that's worth a little tangent to drive home the purpose of an API. It's *always* the API's job to maintain its own data integrity. You shouldn't *ever* be able to do anything in the client code that puts your API data in a weird or broken state. For example, never leave it up to your client code to know that if you delete a list you also have to go delete all the items in the list. That's the API's job.

Continuing… calling the `trash` method sets a local state property `removing` and then calls the API method that sends the command to the server to delete the task (in this case via WebSocket, but the transport is irrelevant).

But the cool thing is, that's *it*. That's all we have to do when we want to delete a widget. You simply have to look up that widget's model and call `.trash()`.

Nowhere in this code do you see anything about removing the item from DOM.

That happens when we get confirmation from the API that the task was removed. It is then removed from the collection, which triggers a `remove` event on the collection and the view (which represents the DOM, as described in the next chapter) listens for `remove` events and plucks that list item out of the DOM. It may sound a bit complex, but only in that you have to describe all those relationships. Once you have it's beautifully simple.

Assuming we've got a view that represents that model, the view would have a click handler like this:

```
var Backbone = require('backbone');
```

```
module.exports = Backbone.View.extend({
  // Our events hash (explained in the next chapter)
  events: {
    'click .delete': 'handleDeleteClick'
  },
  // Our handler simply calls "trash", nothing more
  handleDeleteClick: function () {
    this.model.trash();
  },
  ...
})
```

Alternately, you can simply open the JS console in your browser and type:

```
> app.currentTeam.tasks.get('someId').trash();
```

And you'll see that everything still happens exactly the same as if you had clicked to delete. The task will be deleted on the server and removed from the DOM.

Using models for everything

As the app becomes more complex the failure to store all state in one (and only one) place in your app will be the source of the sorts of bugs that drive you to give up development and take up gardening.

So, what do I mean by storing *all* the state in your app? It's quite easy. If you find yourself checking whether something has a class or not, and using that to determine a course of action, you're doing it wrong.

There are two simple rules:

1. All input, whether from the user or from an API, *never* does anything other than call a method or update a property of your models.

2. Always use your models as the "source of truth" in your app. Never

"look up" state information anywhere other than your models.

Applying this approach to our widget example

Let's think about the data first, before we think about the behavior. These items in the list represent something. Let's model *that* before we think about how they'll be presented. Let's just make a collection of models representing the items in the list.

models/widgetCollection.js

```
var Backbone = require('backbone');
var WidgetModel = require('./models/widget')

// Our main export from this module (just the collection)
module.exports = Backbone.Collection.extend({
  // Specify the model type for this collection
  model: WidgetModel,
  // The RESTful API URL representing this resource
  url: '/widgets'
});
```

models/widget.js

```
var HumanModel = require('human-model');

module.exports = HumanModel.define({
  // We give our model a type
  type: 'widget',
  // Define properties, these are the ones
  // that live on the server-side
  props: {
    id: ['string', true],
    widgetType: ['string', true, 'dooDad']
  },
  // Session properties are just like props but
  // exist for the purpose of storing client-side
  // state.
  session: {
    selected: ['boolean', true, false]
  }
});
```

At this point we can do something like this in our application launch code:

```
var WidgetCollection = require('models/widgets');

// Assume this is the app's entry point
module.exports = {
  blastoff: function () {
    // Create our one global that holds the app
    window.app = this;

    // Attach our widget collection here
    this.widgets = new WidgetCollection();
    // Assumes you've got things set up so
    // this will do an AJAX (or some other type) call
    // and populate the collection.
    this.widgets.fetch();
  }
};
```

So now we've got a representation of that list of widgets that assumes nothing about how it's going to be used.

Stop for a second and think about what that does for us when requirements change or even when we go build a second application on the same API. Nearly *all* the model code will be re-usable with zero changes. It simply represents the state that is available in the API, which is the same no matter what the interface looks like.

Also, think about this in a team environment. Someone can be working on writing models and making sure they get the proper data populated from the API while someone is building the clientside router and page views that include and design "static" versions of page elements that will be rendered by models once the API is hooked up. Because they're all in separate files you won't step on each other's toes and merging the combined code in git won't result in any major merge conflicts.

Just imagine the sort of impact this has for a team to be able to work in parallel and to write code that doesn't need to be thrown away the minute someone wants to change the layout of the app.

In fact, that basic model layer and API synchronization can be created

before we even have a final app design.

Model alternatives

In order to provide observability, models generally provide some sort of event registration system and a way to set and get some “protected” attributes.

For a long time, I used Backbone models for everything. The code for them is quite simple and readable (YES!); they’re also flexible and easy to use. Also, I’m generally a big fan of Backbone’s principles and structure.

Yet, you’ll notice the examples all use `HumanModel` instead of `Backbone.Model`.

Despite my love for Backbone, a few things finally drove me to creating HumanModel:

1. Readability

If the models are the core of our application (as they should be), someone should be able to open the code for the model and *read* what properties it stores and what types those properties have. This is *huge* for enabling people to jump in and contribute to a project.

2. Derived properties

So often, the data you get from the server is not in the format you’ll want to present it. The classic example is first and last name. Most likely they come as separate fields from the API, but in reality, most places you’re going to present a user’s name in the app will be in the format: `firstName + ' '`

`+ lastName`. In Backbone you'd perhaps create a method called `fullName()` that when called, returned that value to you. The annoying thing comes when you want to bind that value to some location in the DOM. You have to listen for changes to either `firstName` or `lastName` and then call the method again and put the result into the DOM. There are two things I don't like about this:

1. It *feels* like `fullName` or even just `name` should just be accessible in the same way as first or last name. Why can't I just go `user.name`?

2. I want to be able to listen for changes in one place. So, instead of `model.on('change:firstName change:lastName', doSomething)` it seems like I should be able to just listen for changes to `change:name` and have the model be smart enough to know that if either first or last name changes, call that handler too.

3. Direct access to properties

In a large app, you work with models **a lot**. Having to call `get` and `set` everywhere is a bit less than ideal, IMO. ECMAScript 5 (a.k.a. the version of the JS spec available in modern browsers) allows for `getters` and `setters` which means you can actually process simple assignments. This is better illustrated with an example:

```
// Without getters/setters (Backbone Model)
model.set('firstName', 'Henrik');

// With getters/setters (HumanModel)
model.firstName = 'Henrik';
```

What do I mean? You can already set whatever properties you want directly on an object even without getter/setters, right?!

YES! But not in a way that can be observed.

Getters and setters allow us to trigger those `change` events even when properties are set directly:

```
model.on('change:firstName', function () {
  console.log('firstName changed!');
});

// Even when setting the attribute directly the callback
// registered above would still be called.
model.firstName = 'Henrik';
```

Getters and setters give us enormous flexibility, which can be bad. For example, we can make a getter run whatever code we want and return anything whenever we access a property.

```
trickyModel.firstName = 'Henrik';

console.log(trickyModel.firstName);
// We can make this log out *whatever* the heck we want
// despite it appearing to just have been assigned above.
```

Quick note on how to use getters/setters

Be sure to read the warning below, but for those not familiar, it may be useful to have a bit of an explanation of how getters and setters are written. There are two syntax options.

The first is using the `get` and `set` operators directly to define those methods:

```
var myObject: {
  _properties: {},
  get name () {
    return this._properties.name;
  },
  set name (value) {
    this._properties.name = value;
  }
}
```

The second is using `Object.defineProperty()`:

```
var myObject = {
  _properties: {}
};

Object.defineProperty(myObject, "name", {
  get: function () {
    return this._properties.name;
  },
  set: function (value) {
    this._properties.name = value;
  }
});

// There's also a defineProperties (plural)
Object.defineProperties(myObject, {
  lastName: {
    get: function () { ... },
    set: function () { ... }
  },
  fullName: {
    get: function () { ... },
    set: function () { ... }
  }
})
```

Warning!

As you can imagine this power gives you a *lot* of rope to hang yourself with and thus, this capability should be used *very* cautiously.

Some argue, and I can see their point, that using this is too much magic. If that's how you feel. Luckily, in our happy modular world, you can just use plain Backbone models and for many simpler apps, I still do.

However, I happen to think that in the case of models getters/settings can actually make our code more fault tolerant and more readable. But, I *only* use them for model properties and only in predictable ways.

4. Type enforcement

JavaScript, the language is dynamically typed, which is awesome. But we've said we're making our models the *core* of the app. Knowing that a given property is a given type is quite useful for eliminating silly bugs and

protecting ourselves.

Let's compare the two with a simple user model. In Backbone there is no standard way to define a property. Instead, you simply set a value as if it exists and now it does.

```
// Backbone model, no definition needed.
// There *is* no standard way to even
// define the properties it should store.
var model = new Backbone.Model();

model.set({
  firstName: 'Henrik',
  lastName: 'Joreteg'
});

// Now I can get those
model.get('firstName'); // logs out 'Henrik'
```

Simple, elegant, flexible. But, assuming I set these values in some view code somewhere in another part of a large app, how do I know what attributes I have available to me or what they're called?

If I'm hitting an API to get my data and using the resulting data to set attributes on models, I have two options for figuring out what data I'm storing and what data I have available to my views. I either inspect the request to know what properties I'm supposed to have or inspect it in the console at runtime to see what properties my model contains and what their names are.

That doesn't seem very developer friendly.

Just think how much information I'm missing:

1. What properties do I have?
2. What type of values do those properties contain?
3. Can I trust that this property will always contain a value?
4. Is this a property client state or data we got from the server?

5. When I go to update the model on the server, which properties should be sent?
6. Is a property computed from other properties, if so, how do I keep it up to date?
7. Perhaps most importantly, where do I go to find the answers to the questions above?

Sure the following example is silly, but what if I write some stupid code (as we do sometimes, amirite humans?!).

```
// There's nothing stopping you from setting
// the firstName property to be a date object.
model.set('firstName', new Date());
```

Sure, you may be able to keep it all in your head to a point, but what about when a second developer comes and looks at that code? Or what happens when you come back to the code after six months (or even two weeks)? Where do you go to see how the app is structured? You have to go spelunking into views for answers.

I prefer that the model is the explicit documentation on what state is stored.

See how this could be in HumanModel:

file: `models/user.js`

```
var HumanModel = require('human-model');

module.exports = HumanModel.define({
  type: 'user',
  // Our properties from the server
  props: {
    // Here is the shorthand syntax for defining a property:
    // first is type, second is required, last is default value.
    firstName: ['string', true, ''],
    lastName: ['string', true, ''],
    // You can also be even more explicit
    // and pass and object
```

```
    middleName: {
      type: 'string',
      required: true,
      default: ''
    },
    // Or less specific the minimum
    // you need is a type, for example:
    isAwesome: 'boolean'
  },
  // Session properties are defined and work exactly
  // the same way as properties. The difference is
  // they're not sent to the server on save(), etc.
  session: {
    selected: ['boolean', true, false]
  },
  // Derived properties are getters constructed from
  // other information. (You cannot set a derived property, this is intentional)
  derived: {
    // The name of the derived property.
    // In this case referencing "model.fullName"
    // would give us the result of calling the
    // function below
    fullName: {
      // We specify which properties
      // this is dependent on (meaning if they change
      // so does the derived property)
      deps: ['firstName', 'lastName'],
      fn: function () {
        return (this.firstName + ' ' + this.lastName).trim();
      },
      // We can optionally cache the result. Doing this
      // means it won't run the function to return the result
      // unless one of the dependency values has changed since
      // the last time it was run. This feature can lead to
      // dramatic performance improvements over plain Backbone.
      cache: true
    }
  }
});
```

file: `hypothetical_app.js`

```
// Grab our user definition from above
var User = require('./models/user');

// Create an instance of that user model
var model = new User();

// I can now know that it's got a value for firstName
console.log(model.firstName); // prints: ''

// and it's a predictable type
console.log(typeof model.firstName); // prints: 'string'

// and we can't just set it to something else
model.firstName = ['hi']; // <- this won't work and will throw and error

// but we can set it as a string
model.firstName = 'Henrik';
```

```
console.log(model.firstName); // prints: 'Henrik'

// Here's the *awesome* part I _can't_ set a property that isn't defined.
// So if I fatfinger the property name, it won't stick.
// *Note what HumanModel does when calling `set()` with undeclared attributes
// can be configured. See human-model docs for more.
model.frstName = 'Henrik';
```

By enforcing this level of property definitions we always make sure that our models, which is the “backbone” of the app (*wink*), are readable pieces of code that help document how the app is put together.

5. Better handling of lists/dates

Another argument for using getters/setters for models is that it makes it possible to observe change to properties that are Objects. At least when using things like arrays and dates as properties.

Since arrays and dates are Objects in JS, they’re passed by reference.

So, what happens if we want to store a list of IDs as a property of a user?

In Backbone, how would we get a `change` event?

```
var model = new Backbone.Model();

model.set('ids', ['23', '25', '47']);

// If we want to get them and change them
var myIds = model.get('ids');

// If we now change it…
myIds.reverse();

// ...and set it back
model.set('ids', myIds);

// we would never get a change event from Backbone
```

If you understand JavaScript you’ll realize this isn’t a flaw in Backbone, it’s just because JavaScript passes objects by reference.

As a result, when Backbone gets the “set” event it just compares `this.get('ids') === newIds` which will always be true, because you’re comparing the same object not a copy of it.

The same is true with dates. If you get a date object, call a method on it, like `setHours()` and set it back, you’d never get a change event. So you wouldn’t know you need to update your view.

We can solve this with getters/setters in cases where we *know* we want this behavior by forcing the model to always give us a new object when we access the property.

```
var HumanModel = require('human-model');

// Set up a simple model definition
var DemoModel = HumanModel.define({
  props: {
    ids: ['array', true, []]
  }
});


// Then we use it
var model = new DemoModel();

// Get our array
var arr = model.ids;

// Modify it back
arr.push('something');

// This now triggers a change event
model.ids = arr;
```

Summarizing models

Models should contain the following:

1. Properties that we get from the API
2. Properties that we need in order to track client state (selected, etc.)

3. Mechanisms for validating their own data integrity
4. Methods we can call to update or delete corresponding models on the server
5. Convenient accessors (a.k.a. derived properties) that describe or process the properties in some way to allow re-use
6. Child collections (if applicable)

Models should *never* contain:

1. Anything that manipulates the DOM
2. Any DOM event handlers

Views

In the interest of being terribly cliché, views are where the rubber hits the road. It's where your model layer meets the DOM.

Before we get into the details, let's talk a bit about why I believe views are a great pattern. The main thing they give us is a clean way to encapsulate and store all the logic for how your application interacts with the DOM. In fact, it's even more specific than that; we use them to contain all the logic for a *certain element* within the DOM. Each view is responsible for the content, event handling, and updating of a single element and the event handlers in views translate user actions into changes to models.

As I've already alluded to in previous chapters, separating application models and views buys us a *lot* of flexibility. We can change the layout and HTML structure of the whole app without having to change anything about how the app gets, stores or updates its data from an API. So in the same way that CSS helps us clearly separate the styling of a document from the HTML content, views help us separate DOM creation, updates, and events from the model layer in our app.

Another *huge* benefit of views is that they let us keep all event handlers (click handlers, etc.) cleanly bundled with the relevant portion of the DOM. If you've ever tried to build a single page app without views, you'll know that managing large numbers of event handlers tends to be a big source of bugs, memory leaks, and messy code.

There are many tools, frameworks, and approaches to handling this layer – all with varying degrees of magic.

So, continuing our theme of striving for readability and separation of concerns, we want something simple, explicit, and declarative.

Backbone views provide some really great basic patterns for building the view layer:

1. One root element that the view controls, available as `this.el` within the view.
2. One primary model and/or collection, available as `this.model` and `this.collection` respectively.
3. A `render()` method responsible for populating and maintaining that base element with the proper contents.
4. An optional `initialize()` method for any necessary setup.
5. Shorthand way to register DOM event handlers (the `this.events` hash).
6. A way of disposing of the view and any listeners that it registered.

But, they're *quite* basic so in addition, we'll extend Backbone views to enable:

1. Simple templating using our precompiled template functions described in Chapter 8.
2. A simple way to declare model/template bindings.

Introducing HumanView

As I mentioned Backbone views are very limited in scope – quite intentionally so. The following explanation is pulled straight from the Backbone docs:

> Backbone views are almost more convention than they are code — they don't determine anything about your HTML or CSS for you, and can be used with any JavaScript templating library. The general idea is to organize your interface into logical views, backed by models, each of which can be updated independently when the model changes, without having to redraw the page.

Backbone's general approach is to provide some simple components and patterns, and it's up to you to apply them as you wish. This non-prescriptive flexibility is a big reason why Backbone has become as popular as it has.

However, as you start to build more and more apps you find yourself solving similar problems over and over. In pure Backbone projects we found ourselves always creating a `BaseView` that contained a lot of the common helpers and patterns we wanted in all our views, and we used that to build all our views in the app. One day I found myself copying and pasting one of the `BaseView`s from one project to another, and just decided to put in on npm instead.

That's how HumanView was born. It's just a Backbone view that gives us a few additional goodies.

Specifically, it gives us the following:

1. Declarative data bindings.
2. A `.renderAndBind()` method that does several things we want to do on every render.
3. A `.listenToAndRun()` convenience method for binding view methods to model events, while maintaining the view as the context and triggering them right away.
4. A `.renderCollection()` method for rendering a view for each item in a collection within a given container element in the view.

We'll take a look at each of those shortly. But first, let's figure out how we're going to structure our views within the app.

A Hierarchy of Views

As you start to build an application with views, you'll find it makes sense to segment things into subviews. Which raises the question, how do you determine what portions of the app layout to split into subviews?

I generally start with a single main view, that I put in `views/main.js`. The main view has the `<body>` as its root element. It's only rendered once and creates the main layout of the app, often rendering several subviews. It also becomes the logical place to register "global" event handlers for things like keyboard shortcuts or app-wide click handlers.

The layout will vary from one app to the next, but there are typically some ever-present elements that are part of the layout (navigation, etc.) and often I will have some type of main content container that swaps out based on the URL. I typically give that an `id` of `pages` and then render a `PageView` into that container based on the URL.

Here's an example of how a main view might look if we're using

HumanView:

```
var HumanView = require('human-view');
var templates = require('templates');
var NavigationView = require('./navigation');


module.exports = HumanView.extend({
  // Our template function that returns an HTML string
  // this can also just be a string. Template language
  // is irrelevant. It just needs to be a function that
  // takes an argument and returns a string.
  template: templates.main,

  render: function () {
    this.renderAndBind(); // Inherited from HumanView

    // Init and "render()" a subview for a hypothetical
    // navigation view
    this.navView = new NavigationView({
      el: this.$('#mainNav')[0],
      model: this.model
    }).render();

    // It's common practice to return "this"
    // when rendering Backbone views in order
    // to make it possible to assign the result
    return this;
  }
});
```

You will have to make a judgement call on the best way to segment things into manageable, logical containers for your application. Generally, a good rule of thumb is try to encapsulate views by the models you'll use to control them.

For example, let's assume you've got a certain URL that represents a page that should show a list of items. In this case you have a page view that is rendered inside the main views' page container. That PageView would render any headers for that page, as well as a basic list container (a `<ul>` perhaps) for your list of items.

That page would take the collection you plan to render into that container as its `collection` property, then we could use the `renderCollection` method to manage adding/removing individual views (one for each model).

If there isn't a lot of behavior associated with each line item, you may choose to handle the rendering of individual items in the view containing the collection. You'll simply have to make a determination based on how much behavior is associated with each item in the list. If it's fairly behavior-less or log-like (say a chat room, for example) you might want to render them into the container and be done. If it's more interactive, like an tour scheduling app where you're dragging items around, editing them, and there's lots of associated data with each one, then you'll probably want a view to contain the behavior of each item.

Take a look at the associated demo app to see examples of each approach to handling collections.

You can find the app on my GitHub account

Caveat: understanding `this.$`

Inside the example above, in the `render` method, you'll notice that we pass: `this.$('#mainNav')[0]` as the `el` argument for the subview. You may wonder, why not just pass `$('#mainNav')[0]` or even just `document.getElementById('mainNav')`?

Well, you can't assume that the view is attached to the main DOM tree when this method is called. If you haven't yet attached it, the other selector queries wouldn't be able to find the right elements because they're not in the main document tree yet. In fact, often a parent view will call `render()` on a subview as part of its own render method, and then attach the result to the DOM. This is entirely intentional because it's much faster for the browser to create the DOM elements outside of the main DOM tree, only attaching and painting them once.

So, to deal with this problem, Backbone Views create a method named `$` for each view. This method is functionally equivalent to a normal jQuery selector such as `$('.item')`. *But*, it only looks for matches within the view's element. Not only is it faster (because there's less DOM to traverse) but more importantly, it finds the elements that match your selector within the view's element *even if it's not yet been attached to the DOM*.

If that was all a bit too complex, just know that you should generally use `this.$('.yourSelector')` instead of `$('.yourSelector')` when trying to grab elements within a view.

Registering DOM event handlers

In wiring up a view to the DOM, you'll often want to respond to interactions from the user.

Because registering a handler to a particular method in your view and binding it to execute in the context of the view is such a common pattern, Backbone gives us a declarative short way to register all the handlers we'll need for a given view.

This is done through the `events` hash.

It works like this:

```
var HumanView = require('human-view');
var templates = require('templates');


module.exports = HumanView.extend({
  template: templates.widget,
  events: {
    // The event + element: the name of the handler
    'click .delete': 'handleDeleteClick',
    'keyup input.search': 'handleSearchKeyUp'
  },
```

```
  render: function () {
    // This we inherit from human-view
    this.renderAndBind();
  },
  handleDeleteClick: function () {
    this.model.delete();
  },
  handleSearchKeyUp: function () {
    var inputVal = this.$('.search').val();
    this.collection.each(function (model) {
      model.matchesSearch = this.name.indexOf(inputVal) !== -1;
    });
  }
});
```

That events hash is equivalent to doing the following inside the render method.

```
  render: function () {
    // This we inherit from human-view
    this.renderAndBind();
    this.$el.delegate('.delete', 'click', _.bind(this.handleDeleteClick, this));
    this.$el.delegate('input.search', 'keyup', _.bind(this.handleSearchKeyUp, this));
  },
```

But the events hash is less verbose and arguably more readable.

Binding model values to templates

In order to keep our separation of concerns, very rarely do I set style attributes directly from JavaScript. I believe that is a job for CSS. So much of what I do is flip classes based on property values on the underlying model.

Backbone kind of loosely encourages you to just re-render views entirely when something changes. In a lot of cases that's totally fine, but I like only changing the specific thing that needs updating when the underlying model changes. Obviously, this can be a bit more tedious because you have to bind each thing explicitly somehow.

This is where HumanView comes in handy. Much in the same way that we declare event handlers in the `events` hash as described above, we can also declare data bindings of various types in our views as follows:

```
var HumanView = require('human-view');
var templates = require('templates');


module.exports = HumanView.extend({
  template: templates.widget,
  events: {
    'click .delete': 'handleDeleteClick',
    'keyup input.search': 'handleSearchKeyUp'
  },
  // Content bindings mean
  // put the name attribute of the
  // model in this view into the
  // element that matches the
  // '.profileName' selector as text.
  contentBindings: {
    'name': '.profileName'
  },
  // Class bindings work a tad differently.
  // If they're boolean attributes
  // it will add or remove a class
  // of the same name as the property.
  // If the property value is a string
  // it will maintain a class of whatever
  // that string value is on the element.
  classBindings: {
    'selected': '',
    'active': '.container'
  },
  render: function () {
    this.renderAndBind(); // This is what does all the bindings.
  },
  handleDeleteClick: function () {
    this.model.delete();
  },
  handleSearchKeyUp: function () {
    var inputVal = this.$('.search').val();
    this.collection.each(function (model) {
      model.matchesSearch = this.name.indexOf(inputVal) !== -1;
    });
  }
});
```

In this way, you follow a similar style and pattern to Backbone to also specify what properties (or computed properties) you want bound to what DOM.

As an additional bonus, all handlers are registered using Backbone's

`listenTo()` which handles unbinding those handlers when the view is destroyed.

HumanView's convenience methods

.renderAndBind();

The general pattern, encouraged in the Backbone documentation is to use templates to populate the contents of a view's main element. That way, you never have to re-register any DOM event handlers because they're attached to the view's root element. With that approach (which is perfect for some uses) you can just call `.render()` any time anything changes in the model.

If you have a simple view that renders a single model, binding views becomes *very* easy at that point. You simply do something like this:

```
var Backbone = require('backbone');
var _ = require('underscore');
var templates = require('templates');


module.exports = Backbone.View.extend({
  template: templates.user,
  events: {
    'click .myClass': 'myHandler'
  },
  initialize: function () {
    // Register a single change handler for the model
    this.listenTo(this.model, 'change', _.bind(this.render, this));
  },
  render: function () {
    // We simply fill the contents of the current element with
    // the rendered HTML using the model's current attributes each time.
    this.$el.html(this.template(this.model.toJSON()));
  },
  myHandler: function () {
    // Do something
  }
});
```

At this point, any change we make to that model will simply re-render the HTML for the whole thing. Slick, simple, and easy.

However, that's not always what you want, especially in realtime apps where an incoming event could come in and change a model when you're not ready for it. But perhaps a more compelling argument is where you want to use CSS3 transitions and animations. If we want to add a class that triggers a transition, simply re-drawing the whole container won't actually trigger it.

Also, it doesn't quite feel right to me to write templates that only contain the contents of the view element:

```
<h1>My page</h1>
<p>My content</p>
```

It seems more logical to write *the entire* template for that view which also includes the root element itself:

```
<section class="page">
   <h1>My page</h1>
   <p>My content</p>
</section>
```

Because now, just by looking at that template, I can look at it and know what it is without having to know which view is going to use it.

In addition, if I want to include some conditional class or some other property on the root element I can do so declaratively, right in the template along with everything else, instead of having to do it in the render method of the view.

Now, enter `renderAndBind(opts)`. Basic render encapsulates everything you need to do to render the view, while also replacing the entire existing root

element and making sure all the event handlers in your event hash are registered.

It looks for a `template` property of the view, and calls it with the context you hand it.

.listenToAndRun();

Very commonly, when you want to listen to some change on a model, you're often wanting to:

1. Bind the handler so that when it's called, `this` is the view.
2. Run the bound handler once so its effect is applied to the DOM. (This avoids having to duplicate logic in the template that's already in your handler).

`.listenToAndRun()` does both of these for you.

So instead of:

```
...
initialize: function () {
  this.listenTo(this.model, 'change', _.bind(this.doSomething, this));
  this.doSomething();
}
...
```

You can just do:

```
...
initialize: function () {
  this.listenToAndRun(this.model, 'change', this.doSomething);
}
...
```

.renderCollection();

`.renderCollection()` is a lightweight way to render and maintain a collection of models within a container.

It will listen for `add`, `remove`, `sort` events on the collection and shuffle and re-draw views for each model as necessary.

You simply pass it the collection, the subview you want to render each model with, and the set of options you want to pass to the subview, and it handles the rest.

Example:

```
var HumanView = require('human-view');
var templates = require('templates');
var ItemView = require('./item');


module.exports = HumanView.extend({
  template: templates.myPage,
  render: function () {
    this.renderAndBind();
    this.$container = this.$('.myItemList')[0];
    this.renderCollection(this.collection, ItemView, this.$container[0]);
  }
});
```

For more on HumanView, or to contribute and make it better, see the documentation and source on GitHub.

A bit about defining bindings in templates (à la AngularJS, Ractive)

There are tools out there that let you specify in your templates which pieces of information go where in your DOM and then they magically handle the event bindings for you.

When I first started working with Backbone when it was v0.3 I thought I wanted this. Basically, assume you have a template like this:

```
<div>
  <p>Hello {{ name }}</p>
</div>
```

Then you mash that together with your model and then they're magically bound. As a paraphrased pseudo-code-y example:

```
var template = require('compiledTemplateFromSomewhere');
var model = require('someModel');

document.body.appendChild(template(model));

// Then if you changed the model
model.set('name', 'Sue');

// the DOM would be magically updated to be:
/*
<div>
  <p>Hello Sue</p>
</div>
*/
```

This is all fine and good for inserting text into an HTML snippet. But what if what you actually want is a bit of logic, or what you want to bind is another attribute, like a `class`, `src`, `href`? Not a big deal per se, but it starts getting more convoluted and pretty soon you're writing a lot of logic into your templates.

Why is that bad? It could be argued, but I feel like it's the wrong place to read logic. I find `if` statements and functions in JavaScript much easier to follow in JavaScript files with the rest of the logic, than when it's sprinkled into the HTML. That reminds me of old approaches to building dynamic web pages where people would write a DB query at the top of the HTML page within some type of special tag and then loop through the results in the markup below using other special tags.

Mixing of these concerns makes re-factoring and code re-use more difficult because you've got bits and pieces of logic spread out in more places.

Stop sending template engines to the browser! Here's a retrospectively obvious way to create templates that happen to be 6 to 10 times faster.

These days, more and more HTML is rendered on the client instead of sent pre-rendered by the server. So if you're building a web app that uses a lot of clientside JavaScript you'll doubtlessly want to create some HTML in the browser.

How we used to do it

First, a bit of history. When I first wrote ICanHaz I was just trying to ease a pain point I was having…generating a bunch of HTML in a browser is a pain.

Why is it a pain? Primarily because JavaScript doesn't cleanly support

multi-line strings, but also because there isn't an awesome string interpolation system built into JS.

To work around that, ICanHaz, as lots of other template clientside template systems do, uses a hack to make it easier to send arbitrary strings to the browser. As it turns out, browsers ignore content in `<script>` tags if you give them a `type` attribute that isn't `text/javascript`. So, ICanHaz reads the content of tags on the page that say: `<script type="text/html">` which can contain templates, or any other multi-line strings for that matter. So, ICanHaz will read those templates and using Jan Lehnardt's awesome mustache.js it turns each of them into a function that you can call to render that string with your data mixed into it. For example:

This HTML:

```
<script id="user" type="text/html">
  <li>
    <p class="name">Hello my name is: {{ name }}</p>
    <p><a href="http://twitter.com/{{ twitter }}">@{{ twitter }}</a></p>
  </li>
</script>
```

Is read by ICanHaz and turned into a function you call with your own like this:

```
// Your data
var data = {
  first_name: "Henrik",
  last_name: "Joreteg"
}

// I can has user??
html = ich.user(data)
```

This works pretty well and is much cleaner than building HTML strings manually. Clearly, lots of people thought the same as it has been quite a popular library.

Why that's less-than-ideal

It totally works, but if you think about it, it's a bit silly. We're making the client do a bunch of extra parsing and compiling that we could actually just do ahead of time. Of course, doing the parsing and compiling in the browser means that we're having to send the browser a whole template engine that can parse and compile templates too.

How we're doing it now

What I finally realized is that all you actually want when doing templating on the client is the end result that ICanHaz gives you: a function that you call with your data that returns your HTML.

Typically, smart template engines, like the newer versions of mustache.js, do this for you. Once the template has been read, it gets compiled into a function that is cached and used for subsequent rendering of that same template.

Thinking about this leaves me asking, why don't we just send the JavaScript template function to the client instead of doing all the template parsing/compiling on the client?

Well, frankly, because I didn't know of a great way to do it.

I started looking around and realized that Jade (which we already use quite a bit at &yet) has support for compiling as a separate process and, in combination with a small little runtime snippet, this lets you create JS functions that only require a small runtime and not the whole template

engine to render. Which is totally awesome!

So, to make it easier to work with, I wrote a little tool: templatizer that you can run on the server-side (using Node) to take a folder full of Jade templates and turn them into a JavaScript module that you can include in your app and contains a function for each template file. Each template function simply takes a context object and returns a string with those values inserted.

The end result

From my tests the actual rendering of templates is **6 to 10 times faster**. In addition you're sending *way* less code to the browser (because you're not sending a whole templating engine) and you're not making the browser do a bunch of work you could have already done ahead of time.

Clientside routing (if there is such a thing)

One thing you sometimes lose with a browser app is the proper handling of the browser's "back" button and the ability to "deep link" into some specific view of the app.

The good news is we don't actually have to make those tradeoffs. Through the clever use of Backbone's router and HTML5 push state, browser apps can take over the world. Here's how it works…

Same sh*t different URL

From the server perspective, how do we actually "hand control of routing to the client"? Ugh… that's not how the web works, right? The server has to answer actual http GET request when a user types your app's URL in their browser.

So what I mean is simply that you return the same app HTML at multiple URLs.

For example, it doesn't matter if you hit:

https://andbang.com/andyet/chat

or

https://andbang.com/basecamp

Either way, the server will return this HTML:

```
<!DOCTYPE html>
<script src="/&!.js"></script>
```

It may be helpful to think about it as a block of URLs that all just serve the app.

If you're using Express and Node it's quite easy to do.

```
app.get('/other/thing', function () {
  // You could still serve other support pages
  // and simple things at specific URLs.
});

// But then you want some sort of catch all that matches
// the range of URLs you're going to want the single page app
// to be available at:
app.get('*', function (req, res) {
  res.sendFile('sameShit.html');
});
```

At this point finding/fetching the right data and rendering the right view is up to the browser app.

How to deal with clientside routes

And Bang has a task detail page for every task at a URL structure that looks like this:

https://andbang.com/andyet/tasks/47

So, if a user types that URL in their browser the user will see a detail view of that task. Also, if they're somewhere else in the app and navigate to that task view by clicking the task in their list, the page won't reload, but when you look at the URL bar in the browser, that's the page you're on.

Backbone's router is really handy for handling all of that stuff.

But, in order to grasp how this works in practice, we have to talk a bit about the application launch sequence.

3… 2… 1… Blastoff!

Generally there's going to be a fairly specific load sequence you'll want to go through before you're ready to "respond" to the specific URL in your client code.

Typically, that sequence goes something like this:

- Init your main application object. (The "app" global I keep talking about.)
- Attach a few model collections to that app global.
- Init and populate a single "me" object that represents the currently logged in user and stores session specific state.
- Render the application layout view inside the `<body>` tag.
- Fetch any app-wide data that's needed.
- Then trigger the clientside router

Now in code:

```
module.exports = {
  // The main launch function. This is the
  // entry point into your application.
  launch: function () {
```

```
    // Explicitly create a global called "app."
    // Doing this first means it *always* exists
    // if we need to access it from a view.
    window.app = this;

    // Attach some model collections
    this.tasks = new Tasks();
    this.chatMessages = new Messages();

    // Init a 'me' object
    window.me = new Me();

    this.fetchStandardData(function (err) {
      if (err) {
        // Handle errors
      }
      // render the main viev
      app.view = new MainView({
        model: me
      }).render();

      // Start our router.
      // Init our URL handlers and the history tracker
      new Router();
      app.history = Backbone.history;
      // We have what we need, we can now start our router and show the appropriate page
      app.history.start({pushState: true, root: '/'});
    });

  },
  fetchStandardData: function (mainCallback) {
    var self = this;
    async.parallel([
      function (cb) {
        self.tasks.fetch({success: cb});
      },
      function (cb) {
        me.fetch({success: cb});
      }
    ], mainCallback);
  }
}
```

As you can tell there's a handful of things we do regardless of the URL. Once we've got that sorted, then we init our router and start our history tracking (which enables back-button support).

The client router looks something like this:

```
var Backbone = require('backbone');
var TaskDetailPage = require('pages/taskDetail');
var FourOhFourPage = require('pages/fourOhFour');


module.exports = Backbone.Router.extend({
```

```
  routes: {
    '': 'home',
    ':slug/:slug/task/:taskid': 'memberTaskDetail',
    ...
  },

  // ------- ROUTE HANDLERS ---------
  home: function () {
    app.navigate(app.teams.first().chatUrl);
  },

  memberTaskDetail: function (teamId, memberId, taskId) {
    var team = app.teams.get(teamId);
    var member = team && team.members.get(memberId)

    // make sure we found our models or send to an internal
    // 'not found' page.
    if (!team || !member) return this.fourOhFour();

    // We may or may not have the task, so we just pass it in and try to get it from the
view.
    app.renderPage(new TaskDetailPage({
      team: team,
      member: member,
      taskId: taskId
    }));
  },

  fourOhFour: function () {
    app.renderPage(new FourOhFourPage());
  }
}
```

So, each of the routes listed at the top are turned into regexes by Backbone and linked to a handler function.

That function is called with the “arguments,” a.k.a. parameters you specified as being dynamic in your routes.

I typically think of each handler’s job as finding actual clientside model objects and then creating a “page” view that it passes to the application.

The app is responsible for taking that view and rendering it per conventions of the app. Usually we just have a “page view” be a specialized kind of Backbone view that also has a few standard methods for “show” and “hide.” The app controller just calls “show” on the new one and “hide” on the currently active page, and the views add/remove themselves from the application layout’s main “pages” container.

From this point forward we never need to do the launch sequence again. We'll just change the route, then the route handlers will render the appropriate page, and the page will ensure it has (or fetches) the data it needs.

Testing and QA that doesn’t suck (so your app won’t)

The problem/challenge of proper interface QA

We all know we need to test. While in the building phase of a browser app I find that building an extensive test suite isn’t necessarily worth the effort. Interfaces are inherently hard to test in that the variability of human input is part of what makes a good test for an interface.

Also, there are things that are just a bit hard to test. For example, if you support drag and drop actions in your interface it’s not so easy to write a test that properly validates that it works.

Then there’s the “problem” of CSS changes. The DOM can be in perfect order but if the styles are off things can look quite broken.

Some people build really elaborate QA systems that load their app into a headless browser and takes screenshots that are compared against

reference images, etc. But the amount of effort and setup required for that is simply not practical in most cases.

There are also tools like Selenium that will script a browser for you, but it's a whole lot of work and setup, and then every time you want to change something, if your tests are too specific they'll need to be constantly updated. And if they're too general they'll miss stuff.

While headless browser testing is a really cool idea (PhantomJS, etc.), it doesn't really help you know how your app works in other browsers.

Ultimately, I don't believe you can actually do proper testing of an interface without a human.

So, there must be a balance that can be struck between human approval and oversight and taking advantage of the things computers are good at like process, consistency, and automation.

Meet the SpaceMonkey

So we've been builing one, we're calling it SpaceMonkey. There wasn't anything out there that did this in a way that was browser-agnostic, simple, and blended automation with human approval.

We've still got a lot of work to do on it to make it awesome, but it's included in the app generator and it seemed worth mentioning nonetheless.

It uses QUnit, the clientside testing tool made by the jQuery team. But rather than run simple programmatic tests. It loads your app into an `<iframe>` and walks you through the QA process that you've defined for it. You can automate high-level tests like filling in and submitting forms, etc. But, in

addition, at the times you've specified it will stop and ask the user for visual verification or to perform some user action such as a drag-and-drop interactions or scrolling-related features that may be hard to test programatically. So some of the unit tests are considered a pass or fail based on a human's answer to "did it work?"

This approach makes it possible to define an explicit set of app interactions to test. So it's like a partially automated QA checklist.

Testing in other browsers will just involve running through the test sequence in other those other browsers. Using a service like Browserling or BrowserStack lets us walk through this process in whatever browsers we're targeting.

SpaceMonkey is still under heavy development, but a `clienttests` folder and sample QA workflow is included in the app generated by the humanjs app generator.

You end up writing tests that look like this:

```
// Load the app into our iframe
monkey.loadApp('/', {
    height: 500,
    width: 600,
    bugUrl: 'https://github.com/henrikjoreteg/humanjs-sample-app/issues/new'
});

// This is a normal QUnit test
test('basic app experience', function () {
    // Here we start a chain of interactions
    monkey
        // We can log out messages to the console
        .log('starting')
        // Wait for things to appear
        .waitForVisible('#pages .page')
        // Ask for user confirmation
        .confirm('The app loaded to the home page.')
        .confirm('The "home" nav tab is active')
        // Navigate to different urls
        .goToPage('/collections')
        .confirm('Collection demo page visible')
        .confirm('List of people are visible each with avatars')
        // We can also provide a set of instructions of something
        // that is about to happen. And instruct the user to look
```

```
        // for specific behaviors.
        .instruct('Five users will be added. Please ensure.', [
            'each one is added at the bottom of the list',
            'each has a readable name an avatar'
        ])
        // Simulate clicks
        .click('button.add')
        .click('button.add')
        .click('button.add')
        .click('button.add')
        .click('button.add')
        .confirm('Everything look ok?')
        .confirm('I can visually re-arrange them by pressing, "shuffle"')
        .confirm('I can hit reset and they disappear, and fetch and they come back.')
        .confirm('I can delete them by clicking "delete"')
        .goToPage('/info')
        .confirm('Info page is visible')
        // Call this when done
        .destroy();
});
```

To see it in action:

1. Install the app generator using npm: `npm i humanjs -g`
2. Run the generator and answer its questions: `humanjs`
3. Run the generated app and visit `http://localhost:3000/test` in a browser

As I mentioned, we still have a lot of work to do in making SpaceMonkey a well-structured standalone library. But it seemed useful and functional enough to include.

App settings and configuration

The problem

App configuration and environment settings are always a PITA, right? Especially when you've got configuration settings you want to pass to your single page app. If we've structured everything as building a "static" js file that gets sent, then how do we pass dynamic values into it for things like configuration?

We solve this problem with a simple approach and two specific tools:

- getconfig
- clientconfig

They're in no way dependent on each other but they do play nicely together. (Both are on npm, by the way).

getconfig

First, getconfig. This is a tool for configuring Node web apps. It follows the same assumptions as Express does, in that it looks to an environment variable (NODE_ENV, reference) to determine the mode in which it should run.

It simply uses that to look for a JSON config file that matches the name of the environment. For example: dev_config.json or production_config.json. It defaults to dev if it doesn't find one.

Then, from your Node app you just require "getconfig" and access settings directly on the resulting object, which will have pulled it from the correct config file. Super clean/simple:

```
// We just require the module
var config = require('getconfig');

// Which actually returns our environment-aware config
// from the corresponding config file.
config.mySetting;
```

It just pulls the right config from the right file, no "if" statements, no mess.

clientconfig

On the clientside we do something very similar.

```
var config = require('clientconfig');

config.mySetting;
```

This time, instead of looking for an environment variable, the module looks for a cookie called “config” that contains a JSON encoded config object. It immediately reads, then destroys that cookie (to avoid the overhead of sending it around on any subsequent requests).

Clientconfig reads and returns a “frozen” config object (using ES5 Object.freeze, if available) with our settings. So, it’s basically the equivalent developer experience as getconfig, but for the client.

Using them together

In combination, it gets pretty slick:

Say we have a dev_config.json that looks like this, on our server.

```
{
  "myProjectSetting": "someSetting",
  "clientAppSettings": {
    "apiUrl": "https://api.andbang.com",
    "trackMetrics": false
  }
}
```

*note the “clientAppSettings” section.

Now, in our main server file where we handle the requests for single page apps, we then add on our config cookie with the data from getconfig using a very simple piece of middleware.

```
// Set up our app and require getconfig
var app = require('express')(),
    config = require('getconfig');

// Build a tiny middleware function that sets the cookie
var configMiddleware = function (req, res, next) {
  res.setCookie('config', JSON.stringify(config.clientAppSettings);
  next();
};
```

```
// In the code that serves our singlepage app, use the middleware
app.get('/app', configMiddleware, function (req, res) {
  res.send(clientApp.html());
});
```

Just like that, we've got a simple unified config file on our server where we can store all environment specific settings for the whole app. It becomes very painless to pass settings to our clientapp based on our environment without having to think about it.

The other benefit of this approach is that it lets us keep our clientside JS app as a completely static file. We don't have to do custom builds that write in settings or anything while still avoiding additional HTTP requests to fetch settings.

Security caveats

Please avoid if sending any valuable settings across the wire this way, especially if you're not using https. Cookies are just http headers, so it's best to assume that this is not secure information and thus, should only be used for non-sensitive data.

Caveats/Gotchas

Function bindings

The most common thing I see when teaching people JavaScript, even people who have been working with jQuery for a long time, is understanding how function execution works in JavaScript.

There are 4 ways to call a function in JavaScript:

1. As a stand-alone function:

```
var myFunction = function () {
  console.log('"this" is', this);
};

myFunction(); // Will log out the `window` object (or global in Node)
```

2. As a property of an object:

```
var obj = {};
obj.myFunction = function () {
  console.log('"this" is', this);
```

```
};

obj.myFunction(); // Will log out the 'obj' object

// Now here's where it gets tricky (continuing the same code as above)

var myFunc = obj.myFunction;

myFunc(); // What will this log out as its 'this'?

// the answer is the `window` object
```

3. Using call

```
var myFunc = function () { ... };

// call with a specific context and any number of arguments
myFunc.call({any: 'object'}, 'someArgument', 'someOther');
```

4. Using apply

```
var myFunc = function () { ... };

// apply an array of arguments
myFunc.apply({any: 'object'}, ['someArgument', 'someOther']);
```

So the question is why?

In JavaScript “this” isn’t magic. It’s just an object. It’s whatever you tell it to be when you’re calling the function. It’s simply the context object for that function execution.

So in the case of the second example where we just do `myFunc();` we’re not giving it anything to use as a context, so it uses the global object “window” because a function body will always have a “this” inside that represents the context of execution.

These are not problems JS developers are used to thinking about when building apps with jQuery. jQuery nearly always hands you the current element as the ‘this’ for event handlers, etc. But…as soon as we start

doing Backbone it trips people up a lot. The following is an example of what I see pretty much every person new to Backbone do:

```
var Backbone = require('backbone'),
    templates = require('templates');

module.exports = Backbone.View.extend({
  initialize: function () {
    // Register a handler so that anytime the model changes,
    // call the render function.
    // THIS WILL NOT WORK!
    this.model.on('change', this.render);
  },
  render: function () {
    this.$el.html(template.thing());
  }
});
```

The problem is that inside the render function, “this” won’t be the Backbone view if it’s triggered by a change in the model. You may say, “Well we’re specifying it as a property of something.” In some ways, yes, you wrote `this.render` but you’re actually just referencing the resulting function and giving the specific function, without context to the event registry.

In fact, what you’re doing is no different than this:

```
// Register a handler so that anytime the model changes,
// call the render function.
// THIS WILL NOT WORK!
var render = this.render;
this.model.on('change', render);
```

So, the render function doesn’t have any context when you just provide a pointer to that function (even though the function may ‘live’ on the view).

So, here’s what you do. You can bind a function to a context before it’s run like this.

```
// THIS will work as expected.
// Backbone's event system takes a third argument for the
// context to execute the function with.
this.model.on('change', this.render, this);
```

This leads into the other two ways to execute a function:

```
myFunction = var myFunction = function () {
  console.log('"this" is', this);
};

var someOtherContext = {
  name: 'blah'
};

// Both of these will log out the 'someOtherContext' object
myFunction.apply(someOtherContext);
myFunction.call(someOtherContext);

// In ES5 compliant (read modern) browsers you can also do this
myFunction = myFunction.bind(someOtherContext);
myFunction(); // "this" will be someOtherContext

// Or if you're using underscore it doesn't matter if you're
// in a modern browser or not. You can just do this:

myFunction = _.bind(myFunction, someOtherContext);
myFunction(); // For the same result
```

That's function binding in a nutshell. It's really just info about how the language works. But it's such a common issue with people who are new to Backbone, or less familiar with JavaScript as a language that I figured it was worth explaining.

Gotchas regarding DOM manipulation in views (they may still be detached)

Another common issue is understanding what `this.$()` does in views.

If you've got a div in your template that looks like this: `<div id="myDiv"/>` and we do this in the render function you'll have a problem:

```
var Backbone = require('backbone');

module.exports = Backbone.View.extend({
  render: function () {
    this.$el.html(templates.myTemplate());
```

```
    // Then you try to access that div like so:
    $('#myDiv').on('click', this.doSomething);

    // ^^ myDiv won't be found! If the root element of this
    // view isn't already attached to the DOM.
    return this;
  }
});
```

What many people don't know is that you can pass a second argument to the jQuery function `$(selector)` that is the DOM tree to look within. So if you did `$('#myDiv', this.el)` in the example above, it would always work.

Backbone tries to make things easy for us, rather than having to do that. Remember to always use `this.$()` instead of just `$()` within views. That's just a helper for passing the view's base element to the jQuery function. It's functionally equivalent to passing `this.el` as the second argument.

Failed Ajax requests

Inevitably with single page apps you have to deal with issues of bad connectivity, or issues of stale data and/or expired sessions.

If we're using RESTful JSON APIs we'll be making requests throughout the application's lifecycle. One approach is to stub out a global error handler for all Ajax requests. jQuery makes this fairly simple: http://api.jquery.com/ajaxError. Often as part of an application's main view, I'll register a handler for global Ajax errors that pops up a dialog to show an appropriate message.

A few closing thoughts

1. We're not done

This book will continue to evolve along with the corresponding tools and techniques. We plan to continue publishing updates to keep the contents current. That's one of the great things about ebooks. We've even built a continuous integration system for the book. Any time we push an update to the master branch for the book repository, our CI server generates new ebooks.

2. Staying up to date

Follow @HenrikJoreteg and @andyet on twitter to hear about updates and changes. We'll be turning the site into a jumping off point for finding related tools and resources.

3. Complimentary Resources

- http://docs.humanjavascript.com – Human JavaScript related documentation site.
- http://resources.humanjavascript.com – Fast searchable directory of hand-picked modules.

4. Open source is a group effort

We're under no illusions that the approaches in here are perfect. All of the tools are open source and hosted on GitHub. If you would like to contribute, please get involved.

5. Feedback on the book

If there was anything that was unclear, incorrect, missing, or could be improved on the book itself, we want to hear about it from you. You can email me directly: henrik@andyet.net with any feedback.

6. Thank you, thank you, thank you

At &yet we believe in the Open Web and invest heavily in building applications and tools that improve the web we love. Your support, through buying the book, helps fund our open source efforts. I hope this has given you some tools and confidence to go forth and build awesomer stuff.

I would argue that the web has greater potential for good than any other human invention throughout history. We're a privileged few, who are fortunate enough to have been born at the right time with the right skills to get to participate in building the web. Our time is short. One of my favorite quotes comes from a poet, Mary Lou Oliver, that Adam Brault introduced me to:

> Tell me, what is it you plan to do with your one wild and precious life?

Please consider using your talents to build things that improve the lives of humans in significant ways. <3