

Introduction

What is a Startup?

Startups are in. At [Yale](#), freshmen inspired by *The Social Network* are signing up for CS courses in record numbers. At [Stanford](#), a classroom of 75 students signed up 16 million users and made a million dollars in 10 weeks. And at Harvard, a kid left for the Valley and... well, you know that story. Polls show that the vast majority of Americans today love entrepreneurship, small business, and free enterprise even if they [disagree](#) about other things. And so the net effect is that Silicon Valley, the geocenter of Startup Land, is the [last place in America where people are optimistic](#). But what exactly are they optimistic about? What is this startup thing anyway?

While different people have different definitions, the general consensus is that a *startup* is a business built to [grow extremely rapidly](#). Rapid growth usually necessitates the use of a technology that invalidates the assumptions of incumbent politicians and businesses. Today startups are strongly associated with Silicon Valley, computer science, venture capital (VC), and the internet. But it was not always so.

Startups in the past

In 1800s America, startups were busy pushing forward the highly nontrivial technologies involved in [oil](#), [steel](#), [pharmaceuticals](#), and [telegraphs](#). In early 1900s America, startups were developing the fledgling [automobile](#), [aviation](#), and [telephone](#) industries. Many similarities exist between these earlier startup eras and the present. For example, the [Wright Brothers](#) began the multi-trillion dollar aviation industry in their bike store, similar to the canonical garage startup of today. And like early internet companies conventionally used some variant of [.com](#) in their names, early car companies tended to bear the last names of their founders. Just as Amazon.com survived and Pets.com did not, Ford and Chrysler survive to the present day, while [Auburn](#), [Bates](#), [Christie](#), and [Desberon](#) did not survive far past the initial Cambrian explosion of hundreds of [startup car companies](#).

One of the most important features of these early startup industries was the initially relatively low cost of capital required to start a business and the wide open regulatory, technological, and physical frontiers. Take [Samuel Kier](#), the first [American](#) to refine crude oil into lamp oil:

By purifying the oil, Kier figured, he could make a cheaper, easier-to-get illuminant than the whale oil in common use then. He built a simple one-barrel still at Seventh and Grant streets, where the U.S. Steel Tower is now. By 1850, he was distilling petroleum and selling it for \$1.50 per gallon. Four years later, Kier enlarged the operation to a five-barrel still that is considered to be the first refinery.

Or [Andrew Carnegie](#):

Carnegie constructed his first steel mill in the mid-1870s: the profitable Edgar Thomson Steel Works in Braddock, Pennsylvania. The profits made by the Edgar Thomson Steel Works were sufficiently great to permit Mr. Carnegie and a number of his associates to purchase other nearby steel mills. . . . Carnegie made major technological innovations in the 1880s, especially the installation of the open hearth furnace system at Homestead in 1886.

Or [Eli Lilly](#):

Colonel Eli Lilly, a pharmacist and a veteran of the American Civil War, began formulating plans to create a medical wholesale company while working in partnership at a drug store named Binford & Lilly. . . . His first innovation was gelatin-coating for pills and capsules. Following on his experience of low-quality medicines used in the Civil War, Lilly committed himself to producing only high-quality prescription drugs, in contrast to the common and often ineffective patent medicines of the day. His products, which worked as advertised and gained a reputation for quality, began to become popular in the city. In his first year of business, sales reached \$4,470 and by 1879, they had grown to \$48,000. . . . In 1881 he formally incorporated the company, naming it Eli Lilly and Company. By the late 1880s he was one of the area's leading businessmen with over one-hundred employees and \$200,000 in annual sales.

To say the least, today it is simply not possible for an individual to build a competitive oil refinery on "Seventh and Grant". The capital costs have risen to billions as the industry has matured. And the permitting costs have risen as well; in fact, gaining the necessary permits to build a new refinery - rather than expanding an existing one - is now difficult enough that the last significant new US refinery was [constructed in 1976](#).

And that is a second major important feature: not only were capital costs low, at the beginning of each of these earlier startup eras, there was no permitting process. As one example: in 1921 [Banting and Best](#) came up with the idea for insulin supplementation, in 1922 they had it in a patient's arm, and by 1923 they had won the Nobel Prize. A timeframe of this kind is impossible given the constraints of today's regulatory environment, where it takes ten years and [four billion dollars](#) to ship a drug. As another example, when aviation was invented, there was no Boeing and no FAA. Thus, from 1903 till the [Air Commerce Act of 1926](#) the skies were wide open without much in the way of rules, with fatal crashes as a common occurrence... and two Wright Brothers could start an [airplane company](#) out of a bike store that at one point [merged](#) with Glenn L. Martin's company (ancestor of Lockheed-Martin) and ended up as the [Curtiss-Wright Corporation](#), the "largest aircraft manufacturer in the United States at the end of World War II" and still a [billion dollar business](#) today.

The initial pioneers in these startup eras thus had time to get underway before competitors and regulations alike started presenting barriers to entry. The messy process of innovation resulted in many deaths from refinery fires, railroad collisions, car explosions, airplane crashes, and drug overdoses. Initially, these injuries were accepted as the price of progress. Over time, however, formal politics arrived in the Wild West, competitors with higher quality arose, and regulations were written that effectively criminalized the provision of beta-quality products. The net result was that the barriers to entry rose sharply, and with them the amount of capital required to challenge the incumbents. Simply from a capital and technology standpoint, it became much more difficult to found a company in your garage to rival Boeing or Roche.

Additionally, collaboration on new rules between large companies and regulators [encoded assumptions](#) into law, making certain features mandatory and banning most minimum viable products. The endpoint of industry maturation was that those few potential startups that were not deterred by economic infeasibility were now legally prohibited.

Conceptually, we can think of this process as a sigmoidal curve (Figure 1) replicated in many industries, where an initial Cambrian explosion of startups eventually results in a few pioneers that make it to the top and a combined set of capital/technological/regulatory barriers that discourage the entry of garage-based competitors.

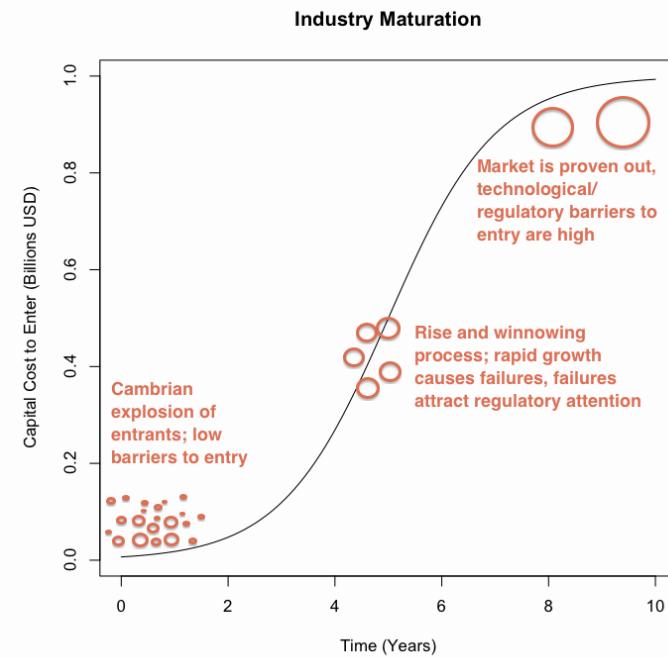


Figure 1: The sigmoidal curve for the development of an industry.

Startups in the present

By the second half of the 20th century, the modern era of venture capital and Silicon Valley was well underway with VCs like [Arthur Rock](#) and Sequoia, semiconductor companies like Fairchild and Intel, and computer companies like Apple. Networking technology advanced as rapidly as the rest of the computer industry, from the first [internet communication](#) to the [founding of Cisco](#), the development of [TCP/IP](#), and finally the invention of the [World Wide Web](#) by CERN's Tim Berners-Lee in 1991. At first glance, these industries might seem no different from previous eras, for the costs of directly competing with Intel's fabs, Cisco's installed router base, or Apple's supply chain are surely in the billions of dollars.

But in addition to these mainly *technological* developments, over the 1989-1992 period two things happened from a *market* perspective that may have fundamentally changed the sigmoidal process. The first was the collapse of the Soviet Union and the worldwide turn towards free markets in Eastern Europe, India, China, and the former USSR. The second was the repeal of the National Science Foundation (NSF) Acceptable Use Policy (AUP), which had banned commerce on the internet. The combination of these two events ushered in the

age of a global internet open for business, and thus the *internet startup*: a business which can start in a dorm room and scale to the entire world, accepting payments and providing services to anyone on the planet without need for natural resources, expensive permits, or human clerks.

</USSR>: Hundreds of Millions Join the Global Market

From the fall of the Berlin Wall in 1989 to the dissolution of the Soviet Union in late 1991, hundreds of millions of people were suddenly freed to start businesses, practice entrepreneurship, and communicate with their neighbors. To give a sense of the shift, in 1987 photocopiers in the Soviet Union were [heavily regulated](#), with proprietors required to padlock them at night to prevent dissidents from mimeographing material critical of communism. From the Los Angeles Times on Oct. 5, 1989:

Soviets Free the Dreaded Photocopier

In what will be a major step toward implementing President Mikhail S. Gorbachev's policy of free-flowing information under *glasnost*...Inspectors from the Interior Ministry will no longer be checking each machine to ensure that it is locked in secure quarters, that complete registers are kept of the documents copied and that the numbers of copies correspond to the tamper-proof counters required on each machine.

Under the elaborate "instructions on procedures of printing nonclassified documentation," adopted in 1974 to prevent dissidents from photocopying their political tracts, then known as *samizdat*, or "self-published" materials, getting anything copied was never quick and never anonymous. . . . "From the first days of Soviet power," the paper added, "even typewriters were put under tight supervision, and that incidentally was abolished only recently."

This sort of total state control over information was characteristic of the entire Eastern Bloc, which is why a pivotal scene in [The Lives of Others](#) revolves around the hero's hidden typewriter: contraband in Communist East Germany.

By 1992, though, newly free Estonians from the former Estonian Soviet Socialist Republic could email anyone in the world from their rapidly burgeoning [internet cafes](#), just a few years after the padlocks came off the photocopiers! With the discrediting of communism, other countries quickly followed suit: under Manmohan Singh, India dismantled the [License Raj](#) and China redoubled its pursuit of market reforms. The fall of the Iron Curtain thus meant that we entered what has been called the second great age of globalization: a highly integrated global market, where businesses can communicate with and serve customers in other countries as never before.

A simple mathematical model for understanding the significance of these events is in [Equation 1](#). Assume that an exchange or communication requires voluntary participation by two parties. In a theoretically perfect globalized economy, any transaction between two parties is legal. However, if the world were a collection of K North Korea-like states with varying shares of world population proportion p_i , with transactions only permitted within countries ($p_1 \times p_1$) and banned between countries ($p_1 \times p_2$), the total proportion of feasible transactions would be:



Figure 2: North Korea vs. South Korea (from globalsecurity.org). North Korea practices a particular variant of communism called [juche](#), which prescribes complete commercial isolation for the country.

$$G = \sum_{i=1}^K p_i^2 \quad (1)$$

Let's call the $G \in [0, 1]$ the globalization coefficient, representing the proportion of allowed transactions, with $G = 1$ corresponding to a perfectly globalized market and $G \rightarrow 0$ as we approach a completely fragmented market of tiny states practicing juche ideology (Figure 2).

At the end of the Cold War, there were roughly 400M in the Eastern Bloc, 1.1B in China, 850M in India, 123M in Japan, 350M in Western Europe, and 250M in the US, with world population around 5.3B. As an extremely crude approximation, assume that the Eastern Bloc, China, India, and (Japan + Western Europe + US + rest of world) could only trade with themselves. Then the proportion of viable transactions before the fall is given as:

$$G = \left(\frac{400}{5300} \right)^2 + \left(\frac{1100}{5300} \right)^2 + \left(\frac{850}{5300} \right)^2 + \left(\frac{5300 - 400 - 1100 - 850}{5300} \right)^2 = .384 \quad (2)$$

While no doubt a very crude estimate, compare this G to the theoretical proportion of 1.0 in a perfectly globalized economy to get some sense of how many new transactions became legal once the USSR, Eastern Europe, China, and India opened for business. Theoretical legality, of course, is different from feasibility - how were these people on different continents supposed to trade with each other?

NSF AUP Repeal: The Internet is Open for Business

During the same period, a debate was brewing at the National Science Foundation. Since the inception of the NSFNET, commerce had been heavily restricted by the [Acceptable Use](#)

[Policy](#) due to fear of widespread spam, pornography, and malware. In part this was because the culture of the internet was oriented towards academic, government, and military users, whose business models depended on tuition, grants, and tax revenue rather than profits; as such they could see (and could foresee) no immediate benefit from the legalization of internet commerce. With market reforms in the air, however, in 1992 Congress [repealed](#) the NSF Acceptable Use Policy, freeing the internet and making it feasible for commercial backbone operators and service providers like AOL to make a profit. Here's [Steve Case](#) (founder of AOL) on the atmosphere at the time:

Next was the pioneering phase (early 80s to early 90s) where a relatively small number of people were trying to build interactive services, but it was an uphill battle. Few people used them (and the Internet itself was still limited to government/educational institutions - indeed, it was illegal to use it for commercial purposes.)

Fascinating! Today, we think of the internet as synonymous with [.com](#), with commerce. Yet this was not an inexorable consequence of technological progress, as fairly sophisticated networking in some form had been around for a while. The market was only ready - the [global internet market](#) only came into being - once the USSR and NSF AUP left the stage of history.

The Key Features of Internet Startups

Many of the trends that began in the early 90s continue to the present day¹, and are particularly salient features of *internet startups* as distinct from other operations:

- *Operational Scalability*: For the first time in history, it is now feasible for an average person to asynchronously trade with someone across the globe at any time of day without special hardware. When you buy a book from an online store, you don't need to leave your house, interact with a teller, or make the transaction during daylight hours. On the other side of the deal, the store can operate 24/7 without regard for zoning regulations, accept transactions without a McDonald's-size army of store clerks, and offer a storefront on every desktop, laptop, and mobile phone on the planet. The theoretical cost-per-transaction (for both buyer and seller) is thus far lower than any physical store.
- *Market Size*: From an international perspective, more than 2.5 billion people in Eastern Europe, India, China, and the former Soviet Union joined the global market for goods, and subsequently the global internet. But even from a purely national perspective, a business in North Dakota can suddenly market its wares over the internet to the entire country, no longer constrained by mere geography (or protected by it). It's still not trivial to scale across borders or states, but the practically addressable market size has never been larger.
- *Generality*: As we will see, software is the most general product imaginable. Software encompasses ideas (news, blogs), entertainment (music, movies), material goods (3D printing, CAD/CAM), communications (email, social networks, mobile phones),

¹While the dot-com boom and bust seemed like a big thing at the time, in hindsight it didn't really interrupt any of these trends. People overestimated change in the short run and underestimated it in the long run, something they are so wont to do that Gartner makes a living off of situating technologies on the [Hype Cycle](#).

transportation (electric cars, self-driving cars), energy (smart grids, adaptive braking), medicine (genomics, EHRs), and more besides. This is why [Software is Eating the World](#) and all products and professions - from watches, maps, and phones to travel agencies, librarians, and photographers - are being [reimagined by software](#). This is also why it's feasible for Google to go from a search engine into maps or from Amazon to go from a bookstore to a server farm: software engineering skills are highly portable, more so than virtually any other skill save mathematics.

- *Low Capital Barriers:* The hyperdeflation of hardware costs begun by Moore's Law shows no end in sight. The result is that for a few hundred dollars, one can purchase an extremely sophisticated general purpose computer capable of developing arbitrary software and communicating with any other computer in the world. The importance of this can be grasped by analogy to the tool industry. When a mechanic needs a special kind of wrench to fix a car, he needs to order that physical object and hope it's in stock. By contrast, when a computer scientist needs a special kind of tool to fix a file format, he needs only to type furiously into a keyboard to fashion that tool out of pure electrons. In this sense, you own the means of production. You have an incredibly powerful desktop manufacturing device on your desk, a kind of virtual 3D printer before its time, whose price is now within reach of even the world's poorest.
- *Low Regulatory Barriers:* Unlike virtually every physical arena, the internet is still mostly unregulated, especially for smaller businesses. In this sense it is not just a technological frontier, but a kind of physical frontier like the New World, the Wild West, international waters, or outer space: a place with relatively little in the way of formal political control, where [code is law](#). One might argue that this is because of the sheer difficulty of regulating it; given the logic of the globalization coefficient G , a given forum today might pull in assets from servers in three countries and comments from a dozen more, making it difficult to determine jurisdiction. However, the Great Firewall of China, the recent wave of political intervention in internet companies, and the recent [SOPA/PIPA](#) and [ITU](#) imbroglios indicate the need for continued vigilance on this point.
- *Open Source:* The open source movement dates back to Richard Stallman's launch of the GNU project in the early 80s, but was turbocharged with the advent of the internet and more recently with the rise of Github. The internet is built on open-source technologies such as Linux, Apache, nginx, Postgres, PHP, Python, Ruby, node, BitTorrent, Webkit, and the like. Open source is of particular importance to this course, and it is worth reading [this essay](#) on open source economics as distinct from both communism and market economics. Briefly, open source resembles the non-commercial gift economies that some of the more well-meaning proponents of communism envisioned, with the all important proviso that said gifts are not only contributed entirely voluntarily but can be also replicated *ad-infinitum* without any further effort on the part of the gift-giver. As such, they are a uniquely scalable means of charitable giving (e.g. the [Open Source Water Pump](#)).
- *The Long Tail:* The sheer scale of the internet enables a second phenomenon: Not only is it possible to address markets of unprecedented size, it is now feasible to address markets of unprecedented *specificity*. With Google Adwords, you can micro-target people searching for arbitrarily arcane keywords. With Facebook Ads, you can reach every single between 27 and 33 with interests in math from small towns in Wisconsin. And

with the right blog you can pull together a community of mathematicians from across the globe to work on **massively collaborative mathematics** in their free time. Never in the history of mankind have so many (l)earned so much from so few.

- *Failure Tolerance:* When a car crashes or a drug is contaminated, people die and new regulations are inevitably passed. The presumption is often that crashes are due to negligence or profit-driven, corner-cutting malevolence rather than random error or user error. Yet when Google or Facebook crashes, people generally yawn and wait for the site to come back up. This colossal difference in the penalty for failure permeates every layer of decision-making at an internet company (viz. “[Move Fast and Break Things](#)”) and sharply differentiates the virtual world from the physical world. Moreover, as computer power grows, we can now simulate more and more of the world on the computer via CAD/CAM, virtual wind tunnels, Monte Carlo risk models, and stress tests, “dying” virtually 1000 times before hitting enter to ship the final product.
- *Amenable to Titration:* In addition to all of the above features, it is possible to build a hybrid business in which some aspects remain constrained by the physical world with aspects of the internet startup titrated in to the maximum extent possible. Arguably this trend started very early with Amazon, which combined a virtual frontend with an all-too-physical backend, but it has recently become more popular with companies like AirBnb, Counsyl, Grubhub, Lockitron, Nest, Taskrabbit, Tesla, and Uber as examples to varying degrees. The advantage of engaging with the physical world is that market demand is proven, price points are generally higher, and real-world impact is tangible. The disadvantage, of course, is that some or all of the above trends may no longer work to your benefit. The key is to titrate at the right level.

These trends work together synergistically to produce the phenomenon of the *internet startup*. Huge global market potential means ready availability of large capital investments. Low-cost scalability means growing pains; most processes designed for handling X widgets will have increased failure rates at 3X and complete failure well before 10X. So sustained meteoric growth is allowed by failure tolerance, as the site *will* experience unexpected difficulties as many more customers arrive. Low regulatory barriers mean that good faith **technical failures** or **rare abuses** don’t result in the business being shut down by regulators, unlike comparable **physical industries**. Generality combined with low capital barriers means that a company that comes up in one vertical (search or books) can go after an allied software vertical (operating systems or ebooks), thereby delaying the plateau of the sigmoidal curve of Figure 1. And generality allows the most successful companies to titrate into a hybrid vertical (automatic cars or grocery delivery) - causing software-based disruption and making the assumption of stable sigmoidal plateaus in other industries more questionable.

Has the internet fundamentally changed the game? Perhaps the combination of an unregulated (unregulatable?) internet, the low capital barriers of pure software, and the incredible sizes of the global internet market will extend the Cambrian explosion period indefinitely, with software winners moving on to disrupt other markets. That’s the bet on continued disruption.

The alternative thesis is that the ongoing financial crisis and possible double dip will affect these trends. Perhaps. One possibility is that politicians begin to exert more centralized control over their citizens in response to fiscal crisis. Another possibility is that the breakup of the EU and a concomitant decline in relative US economic strength will contribute further towards decentralization and a multipolar world. Yet a third possibility is that the recent NSA

scandal makes it harder to scale an internet company, due to national-security-motivated [fracturing](#) of the global internet. The key question is whether the internet will change its international character in the years to come, or whether it will prove powerful enough to keep even the mightiest adversaries [at bay](#).

Technological Trends Toward Mobility and Decentralization

All right. We've just talked about the distant past of startups (Cambrian explosions followed by maturity), and the immediate past/present (the rise of a global internet market and the *internet startup*). What does the future look like? Taken as a whole, a number of new technologies on the horizon point in the direction of continued decentralization and individual mobility:

Table 1: Future (top) and current/ongoing (bottom) technological trends in favor of mobility, decentralization, and individual empowerment.

Technological Trend	Effect
Realtime machine translation	Reduces importance of native language
Plug computer, Raspberry pi	Reduces importance of size, location
Augmented reality	Layer software over the physical world anywhere
3D printing	Reduces patent/political restrictions on physical goods
MOOCs	Hyperdeflation of educational costs
YCRFS9	Decentralize entertainment
Seasteading	Facilitate emigration, reduce political constraints
Crowdfunding	Reduce importance of Silicon Valley VC, banks
Pharma cliff, generics	Complete disruption of US pharmaceutical industry
Bitcoin	Reduce reliance on single reserve currency, banks
Sequencing	See your own genome without an MD prescription
Telepresence robots, drones	Reduce importance of physical location
360-degree treadmill	Permits actual holodeck; reduces importance of location
P2P/Bittorrent	Reduce reliance on centralized backbone
Blogs	Reduce importance of national opinion journalism
Microblogs	Reduce importance of network news outlets
Social networks	Friends are now independent of physical location
Laptops, smartphones, tablets	Computation is now independent of location
Video chat	Reduce importance of physical location
Search engines, ebooks	Access all information, anytime, for free
E-commerce	Reduce importance of physical store location

The overall trend here is very interesting. Focus simply on social networks for a moment and think about the proportion of your Facebook friends who live outside of walking distance. Since the early 1800s and the advent of the telegraph, the telephone, airplanes, email, and now social networking and mobile phones, this proportion has risen dramatically. One of the results is a sharp reduction in the emotional cost of changing your physical location, as you no longer fall out of touch with the majority of your friends. In a very real sense, social networks impose a new [topological metric](#) on the world: for many people who live in apartment complexes, you exchange more bytes of information with your friend across the world (Figure 3) than you do with your next-door neighbor.



Figure 3: Facebook’s Map of the World (from [Facebook Engineering](#)). Each node represents a city, and lines represent the number of friends shared between users in each city pair, sampled from about 10 million friend pairs. Lines between cities within the same country are to be expected. But what is interesting are the sheer number of direct lines between individuals in different nations, or faraway cities. Detailed data analysis would likely show more connections between, say, San Francisco, CA and Cambridge, MA than between San Francisco, CA and Victorville, CA - even though the latter is geographically closer and of approximately the same size (100,000 residents) as Cambridge. And today those connections are maintained not on the basis of yearly letters delivered by transatlantic steamships, but second-by-second messages. This is related to the thesis that the next generation of mobile technologies are not location-based apps, but apps that make location unimportant.

As another example, consider the proportion of your waking hours that you pass in front of a user-programmable screen. For software engineers or knowledge workers more generally (designers, writers, lawyers, physicians) this can easily exceed 50 hours per week of work time and another 10 or more of leisure time. And the primacy of the programmable screen is now extending to traditional industries; for example, Rio Tinto’s [Mine of the Future](#) project has launched its first autonomous robotic mine, complete with self-driving trucks and laser-guided ore sorters, fully controllable from a remote station packed with computer scientists and process engineers, with on-site personnel limited to maintenance activities and most of the work carried out on a mobile glass screen thousands of miles away. Again the importance of physical location is reduced.

As the importance of physical location and nationality declines, and the influence of software over daily life increases via ubiquitous programmable screens (from laptop to smartphone to Google Glass) we may well be due to revisit the [Treaty of Westphalia](#). Briefly, since 1648, the presumption has been that national governments are sovereign over all citizens within their physical borders. Challenges to the Westphalian order have arisen over time from ethnic diasporas, transnational religions, and international terrorist networks. But the global internet may be the most important challenger yet.

We are already approaching a point where 70% of your friends live outside of walking distance, where work centers around a mobile programmable screen, and where your social life increasingly does as well. Today you have a permanent residence and set a homepage on your browser. The physical world still has primacy, with the virtual world secondary. But tomorrow, will you find your community on one of these mobile programmable screens and then choose your physical location accordingly? This would require mobile screens capable of banking, shopping, communications, travel, entertainment, work, and more... mobile devices which hold blueprints for 3D printers and serve as remote controls for [physical locks](#) and

machines. This happens to be exactly where technology is headed.

What is Startup Engineering?

Let's step back from philosophy for a bit and get down to brass tacks. We'll return to philosophy when it comes time to think about business ideas. (Incidentally, this sort of rapid ricochet between abstract theory and down-and-dirty practice is commonplace in startups, like the sport of [chessboxing](#).)



Figure 4: Startups are like chessboxing. The sport alternates rounds of chess and boxing. Checkmate or TKO to win. (image from [Wikipedia](#)).

For the purposes of this class, *startup engineering* means getting something to work well enough for people to buy. Engineering in this sense is distinct from academic science, which only requires that something work well enough to publish a paper, effectively presuming zero paying customers. It is also distinct from theory of a different kind, namely [architecture](#) [astronautics](#), which usually involves planning for an infinite number of users before the very first sale. Between these poles of zero customers and infinite customers lies startup engineering, concerned primarily with shipping a saleable product.

Technologies

One of the primary things a startup engineer does is systems integration: keeping up on new technologies, doing quick evaluations, and then snapping the pieces together. Some people will say that the choice of language doesn't matter, as a good engineer can get anything done with most reasonable technologies. In theory this might be true in the limited sense that Turing-complete languages can perform any action. In practice, the right tool can be like the difference between going to the library and hitting Google.

Below is a reasonably complete list of the technologies we'll be using in the course, with alternatives listed alongside. Note that when building a startup, in the early days you want to first choose the best contemporary technology and then forget about technology to innovate on only *one* thing: your product. Building v1 of your product is unlikely to mean creating (say) a new web framework unless you are a company which sells the framework (or support)

like Meteor. This is why we've chosen some of the most popular and best-tested exemplars in each category: outside of the core technology, you want to be as boring and vanilla² as possible until you begin to make a serious profit from your first product.

Table 2: An overview of technologies used in the course.

Type	Name	Notes	Alternatives
OS	Ubuntu Linux	Most popular Linux distro.	OS X, Windows, BSD
IAAS	AWS	Used for reproducible development.	Joyent, Nirvanix
PAAS	Heroku	Used for easy deployment.	Joyent, Nodejitsu
Shell	bash	Ubiquitous and reliable.	zsh, tcsh, ksh
Text Editor	emacs	Customizable with strong REPL integration.	vim, Textmate, nano, Visual Studio
DVCS	git	Distributed version control champion.	hg, fossil
DVCS Web UI	github	Dominant among web DVCS frontends.	bitbucket, google code, sourceforge
Language	Javascript	The Next Big Language	Python, Ruby, Scala, Haskell, Go
Macro Language	Coffeescript	Makes JS much more convenient to write.	Iced Coffeescript
SSJS Interpreter	node.js	Async server-side JS implementation	v8, rhino
Backend Web Framework	express	Most popular node.js backend framework	Meteor, Derby
Backend ORM	sequelize	Most polished node.js ORM	node-orm, bookshelf
Database	PostgreSQL	Popular/robust fully open-source relational db	MySQL, MongoDB
Data Format	JSON	The industry standard for simple APIs	XML, protobuf, thrift
Web Server	node.js	node on Heroku is reasonably good as a server	nginx, Apache
Dev Browser	Chrome	Chrome Dev Tools are now the best around	Firebug, Safari Inspector
Scraping	phantom.js	Server-side webkit implementation in JS	webscraping.py, wget
Frontend CSS Framework	Bootstrap	Most popular CSS framework	Zurb Foundation
Frontend JS Framework	Angular	A little simpler than Backbone JS	Backbone, Ember, Knockout
Deploy Target	Mobile HTML5	Ubiquitous, forces simplicity, future-proof	Desktop HTML5, iOS, Android

Over the course of the class, we'll introduce the developer tools first (unix, command line), then discuss the frontend (HTML/CSS/JS, browser) and finally the backend (SSJS, devops). We'll also discuss how to structure your application using a Service Oriented Architecture to make it easy to silo old code and introduce new technologies on new servers.

Design, Marketing, and Sales

To ship a product a startup engineer needs versatility. If you are a founder, you will need to handle things you've likely never thought about. When you walk into an engineering class, the lights are on, the room's rent is paid, and you need only focus on understanding code that someone has hopefully made as simple as possible for instructional purposes. But when you start a company, you're responsible for calling the electrician to get the the lights working, finding the money to pay the rent, and (by the way) pushing the envelope of new technology that no one else understands. At the beginning, you have no product, so you have no money to hire other people to help you with these things. And since you have no product, no money, and the lights are off, it can be challenging to get people to quit their high-paying jobs at Google to work with you for free. So you'll need to be able to produce a passable logo, design the first brochures, and do the initial sales calls - all while moving technology forward.

²One point deserves elaboration. If we believe in “boring and vanilla”, why have we chosen so many of the newer Javascript technologies rather than using something older like Django, Rails, or even Java? The short answer is that technology stacks evolve rapidly. At the inception of a company, or a new product, you will gain maximum advantage from selecting the best contemporary technology, one that is clearly rising but may have some rough edges, what Steve Jobs would call a “technology in its spring”. Today, that technology is arguably a full-stack JS framework based on node.js, making heavy use of AWS and Heroku. In 2-3 years it will likely be something different. But once you pick out the elements of your technology stack and begin developing your product, relatively little added value comes from radical shifts in technology (like changing frameworks), and switching technologies can become an excuse for not working on your product.

Now, if you are the first engineer, or one of the first engineers at a startup that has scaled up, you won't need to handle as many of these operational details. But you will still greatly benefit from the ability to handle other things not normally taught in engineering classes, like the ability to design, market, and sell the product you're building. We will thus cover these topics in the early lectures on market research and wireframing, in the later lectures on sales and marketing, and in asides throughout.

Why Mobile HTML5 for the Final Project?

The focus of the final project is on the use of [responsive design](#) to target mobile browsers (like Safari on the iPhone or the Android Browser) for a simple mobile web application, rather than some other platform (like a native Android or iOS app, or the desktop browser). The reason is several fold.

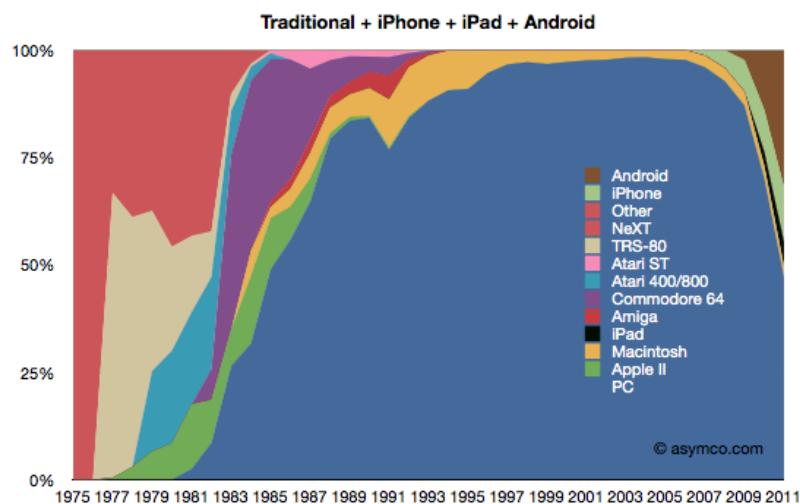


Figure 5: The Rise and Fall of Personal Computing (From [asymco.com](#))

- *Mobile is the Future:* First, anyone involved even tangentially with computers should know that mobile clients are now vastly outselling PC clients (Figure 5), a trend that will only accelerate in the years to come, with the proliferation of a host of new clients beyond phones and tablets including wearables, signage, kiosks, Google Glass, plug computers, drones, and appliances.
- *Responsive Design gives Desktop UI for free:* Second, with a modern responsive design CSS framework like [Twitter Bootstrap](#), it's not that hard to simultaneously target a mobile HTML5 client like a smartphone/tablet browser while also building a desktop app. The rule of thumb is that you design for the mobile interface and limited screen real estate first, and then add more bells and whistles (like larger images) for the desktop site. Over time, you might want to actually do a dedicated mobile and desktop site (compare [m.airbnb.com](#) to [airbnb.com](#)), but responsive design allows you to get an app out the door rapidly and ensure it renders reasonably on most popular clients.
- *Location-Independence increases Scope:* A large class of mobile applications includes many that are location-dependent, like Uber, Foursquare, or Google Maps, which take

your GPS coordinates as a fundamental input. However, as the mobile space evolves, another large class of apps will be those meant to make your current location *unimportant*, e.g. ultra-convenient shopping or banking on the go. Thinking about mobile apps that increase people's location-independence should significantly increase your perceived scope for mobile.

- *Simplicity*: Because mobile screens have greatly reduced real estate, and because users on the go generally don't have the patience for complex input forms, mobile apps force simplicity. This is good for a 4-5 week class project.
- *Ubiquity*: By building an HTML5 app intended for mobile Safari and the Android browser, you will as a byproduct get an app that works in the browser and on most tablet and smartphone platforms. You will also have an API that you can use to build browser, iOS, and Android apps, should you go that route.
- *JS is the Future*: Javascript (aka JS) is truly the [Next Big Language](#). After the [trace trees](#) breakthrough in the late 2000s, fast JS interpreters became available in the browser ([v8](#)), on the server ([node.js](#)), and even in databases ([MongoDB](#)). [JSON](#) as a readable interchange format is likewise ubiquitous, and "[everything that can be written in JS, will be written in JS](#)". Probably the single best reference to get started in JS is [Eloquent Javascript](#).

All right. We now have a sense of what startups and startup engineering is about, and the technologies we'll be using. Time to start coding.

Interactive Start

Your First Webapp

Let's get started. Our first goal is to get you set up with a basic development environment and then get a simple page online by following these steps:

1. Set up Google Chrome and your terminal, and sign up for Amazon Web Services (AWS), Github, Heroku, and Gravatar.
2. Initialize and connect to an AWS cloud instance.
3. Clone code from github.com/heroku/node-js-sample, create a new Heroku app and get your page live on the web (Figure 1).

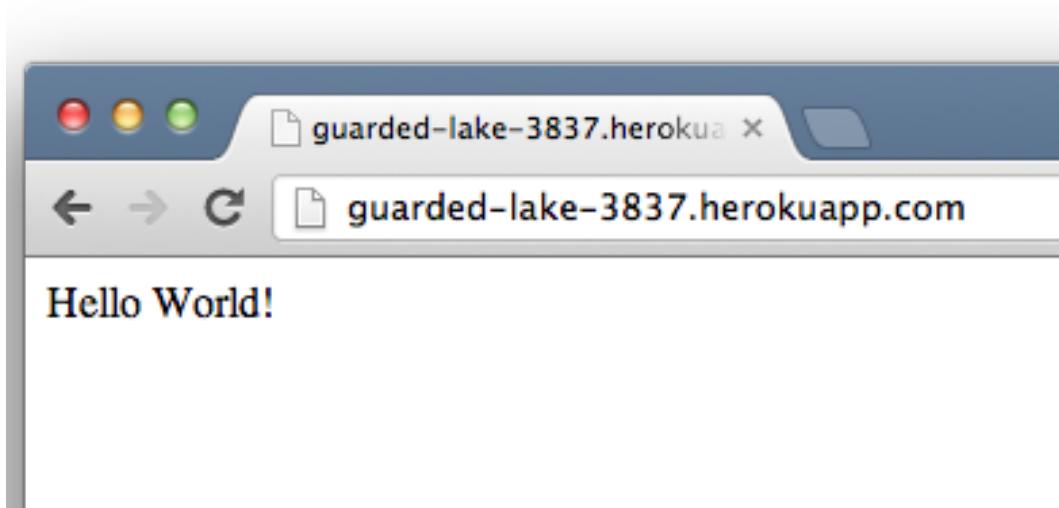


Figure 1: Here is what your final goal is: a simple web page up and live on Heroku.

The general concept of how all these services fit together is illustrated in Figure 2. Even if you don't know what all the terms mean, just follow along with the screens in this lecture. We'll review things more formally once you get your simple site online.

Startup Engineering

Dev and Deployment

Here's how your local computer, AWS, Github, and Heroku all fit together to deploy a simple site.

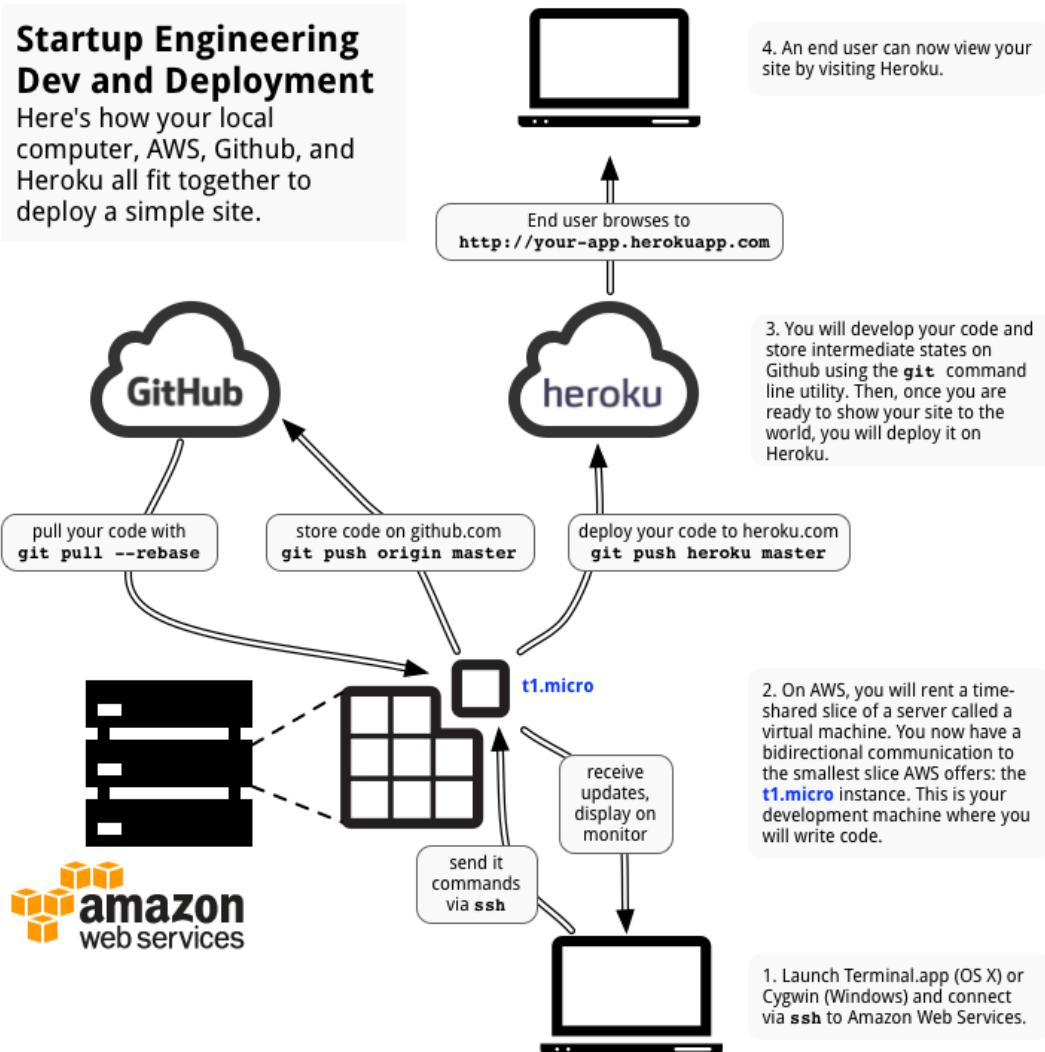


Figure 2: Here's how the pieces we set up today will fit together. During the class, you will remotely control an AWS virtual machine from your laptop and use it to edit code, periodically storing the intermediate results on Github. Once you are ready, you will deploy your code to Heroku for viewing by the world. The reason we recommend AWS rather than your local computer for development is that using a *virtual machine* means you can easily *replicate* the development environment. By contrast, it is much more time consuming to get another engineer set up with all the programs and libraries on your local computer. For this lecture, you won't be pushing code to github, but you will be pulling it from github.com/heroku/node-js-sample.

Setup and Signup

Install Google Chrome

Begin by installing the [Google Chrome web browser](#) and the [User Agent Switcher](#) extension, which you will use to make your browser mimic a smartphone or tablet. Chrome has a number of advanced tools for debugging websites which we will use later on in the class.

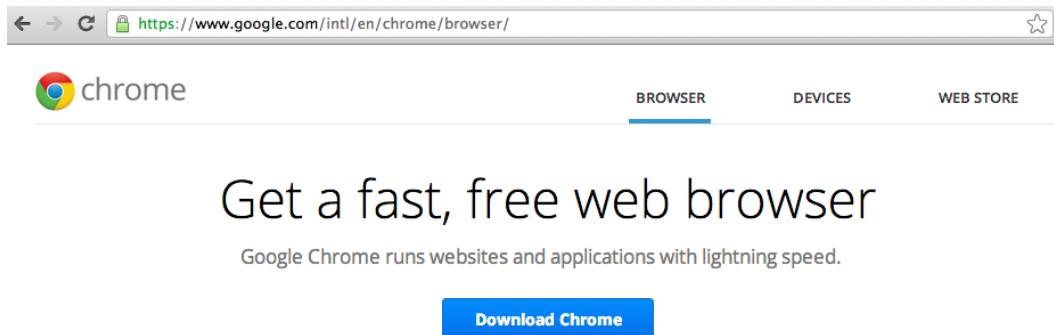


Figure 3: Install [Google Chrome](#).



Figure 4: Install the [Chrome User Agent Switcher](#), to mimic smartphones/tablets.

Set up your terminal

A [command line interface](#) (CLI) is a way to control your computer by typing in commands rather than clicking on buttons in a [graphical user interface](#) (GUI). Most computer users are only doing basic things like clicking on links, watching movies, and playing video games, and GUIs are fine for such purposes.

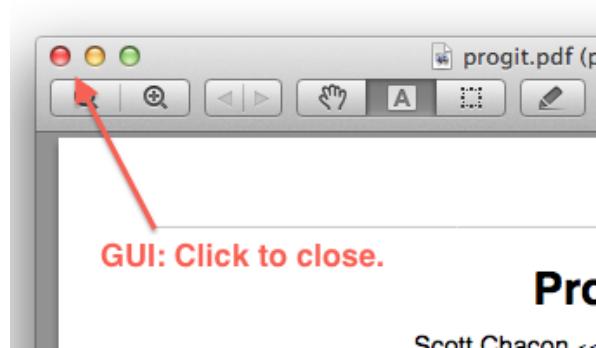


Figure 5: Closing an application via the GUI requires a click.

```
[balajis:~]$ ps xw | grep Preview
2822 ?? S 1:12.53 /Applications/Preview.app,
_0_376924
7766 s006 S+ 0:00.00 grep Preview
[balajis:~]$ kill 2822    CLI: Type "kill" to close.
[balajis:~]$
```

The terminal window shows the command "ps xw | grep Preview" being run, listing two processes: a Preview application and a grep process. The line "2822 ?? S 1:12.53 /Applications/Preview.app," is highlighted with a red rectangle. Below it, the command "kill 2822" is entered, followed by the instruction "CLI: Type "kill" to close." The prompt "[balajis:~]\$" appears again at the bottom.

Figure 6: Achieving the same task with the CLI by running `kill`.

But to do industrial strength programming - to analyze large datasets, ship a webapp, or build a software startup - you will need an intimate familiarity with the CLI. Not only can many daily tasks be done *more quickly* at the command line, many others can *only* be done at the command line, especially in non-Windows environments. You can understand this from an information transmission perspective: while a standard keyboard has 50+ keys that can be hit very precisely in quick succession, achieving the same speed in a GUI is impossible as it would require rapidly moving a mouse cursor over a profusion of 50 buttons. It is for this reason that expert computer users prefer command-line and keyboard-driven interfaces.

Much of the class will be spent connecting to a remote cloud computer and executing commands there, rather than executing commands on your local computer. However, to accomplish this you will still want to set up a basic command line on your local computer.

- Mac OS X: Terminal.app

Apple laptops are preferred by many Silicon Valley engineers because they run BSD Unix under the hood. We recommend getting a Mac with an SSD drive if you wish to join a startup in the Valley; it is standard issue at this point. You can gain access to the Unix command line by on a Mac by opening up Terminal.app, located under /Applications/Utilities/Terminal.app

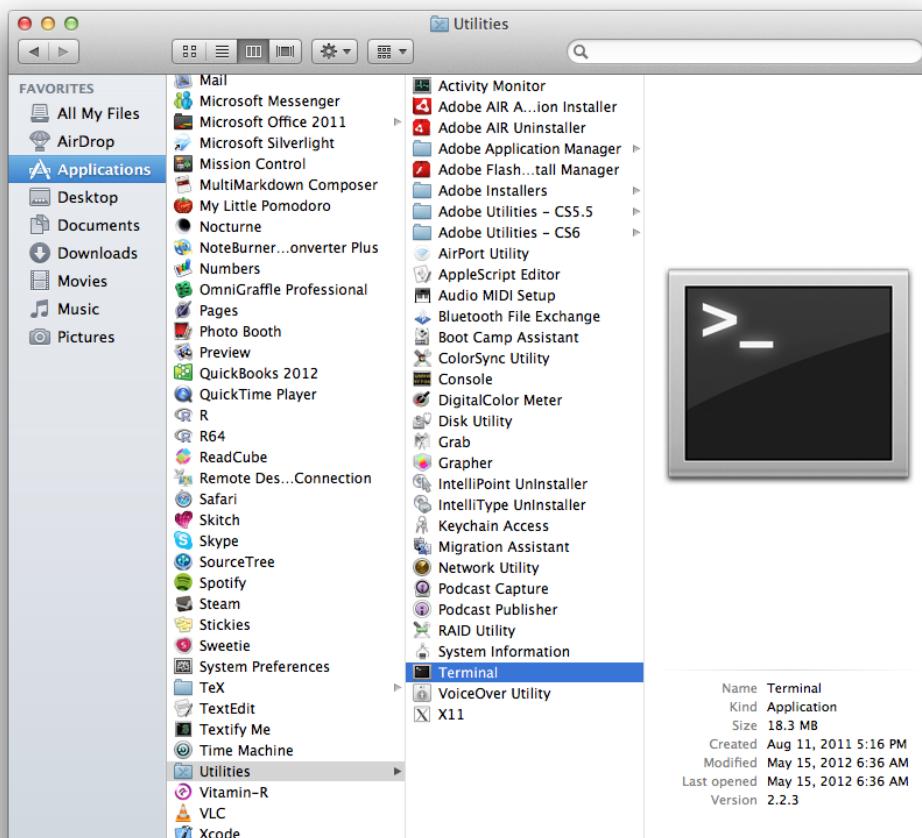


Figure 7: Double click /Applications/Utilities/Terminal.app.

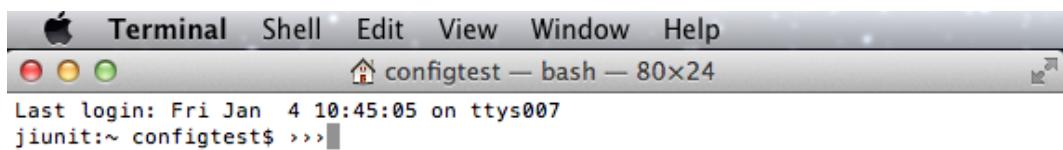


Figure 8: You should see a window like this pop up.

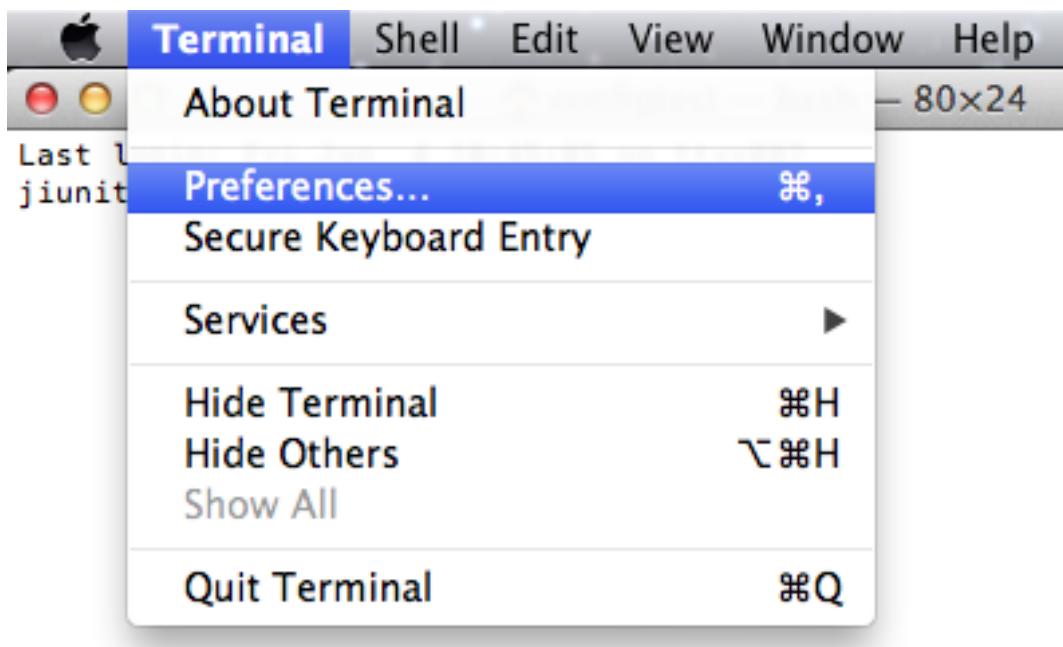


Figure 9: Go to Terminal -> Preferences in the top left.

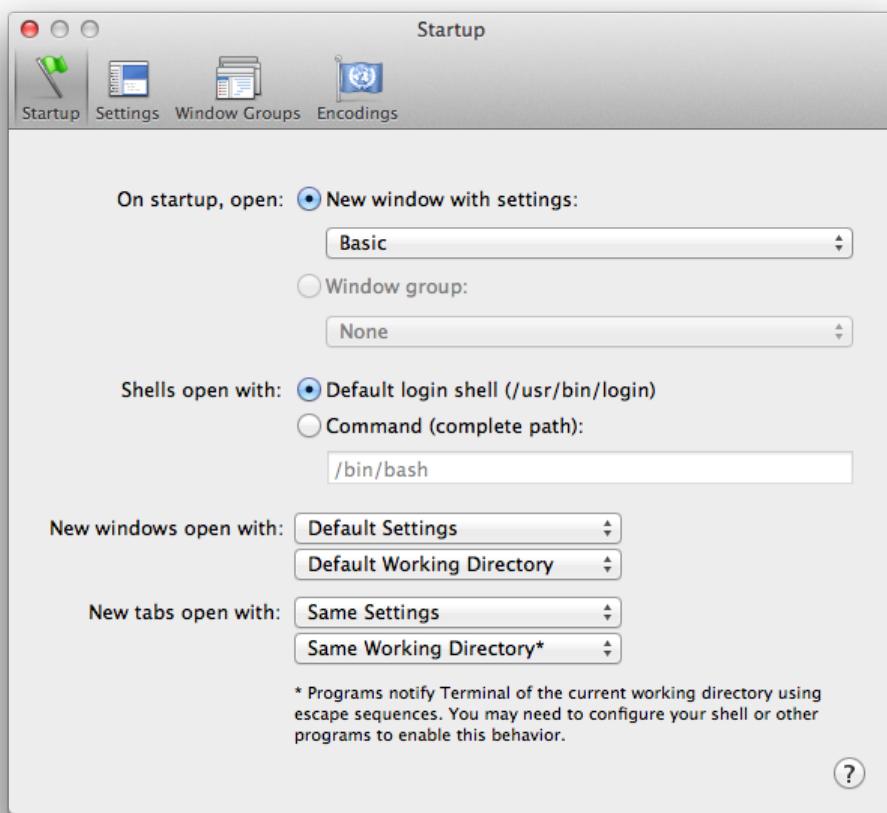


Figure 10: Here's your preference window. There are only a few preferences we need to tweak.

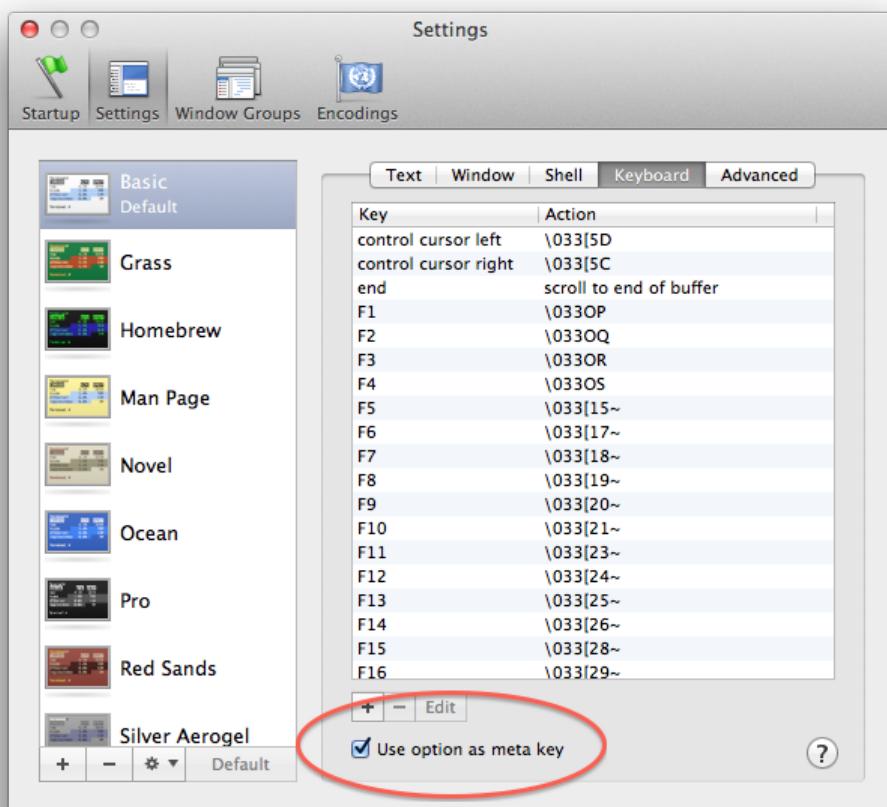


Figure 11: Click *Settings* (in the top row of buttons) and then “Keyboard”. Click “Use option as meta key”. You’ll need this for *emacs*.

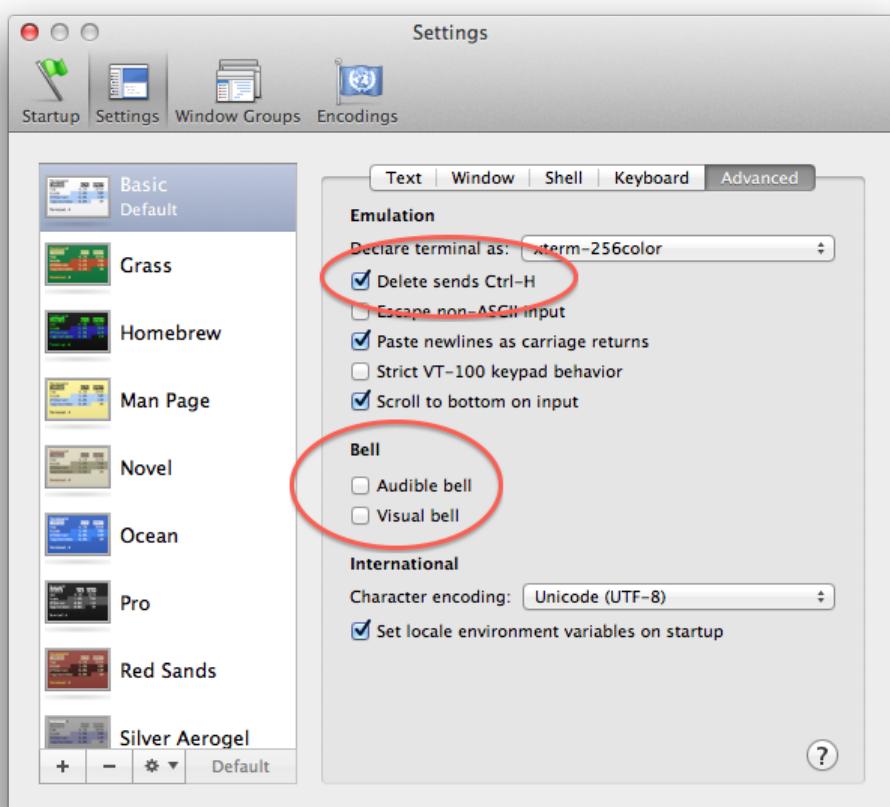


Figure 12: Now click Advanced and set the “Delete sends Ctrl-H” setting. This is useful for Ubuntu.

- Windows: Cygwin

Relatively few startups use Windows computers for their primary development machines, in part because command line support has historically been something of an afterthought. For the many students who still run Windows, for the purposes of this course we are recommending that you use Cygwin to get a Unix-like environment on your Windows computer. Install Cygwin from cygwin.com/setup.exe as shown:



Figure 13: *Cygwin* provides a Linux-like command line environment on Windows.



Figure 14: Download [cygwin.com/setup.exe](http://www.cygwin.com/setup.exe) and double-click it.

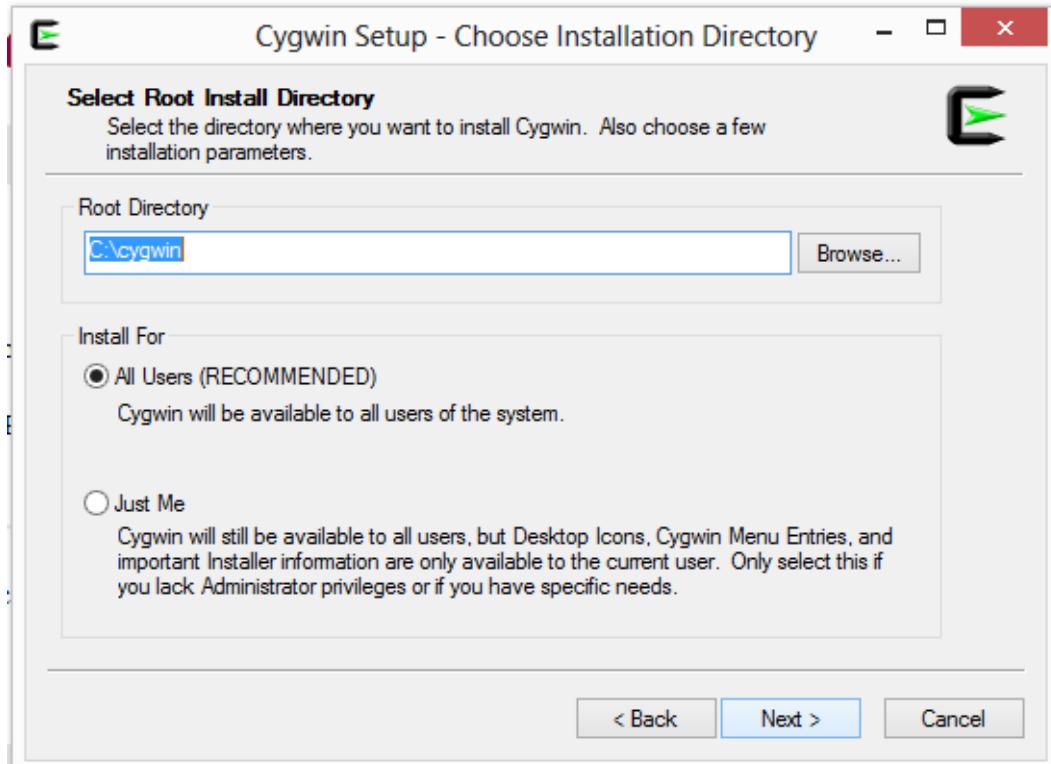


Figure 15: Set up access for all users in the default directory `C:\cygwin`

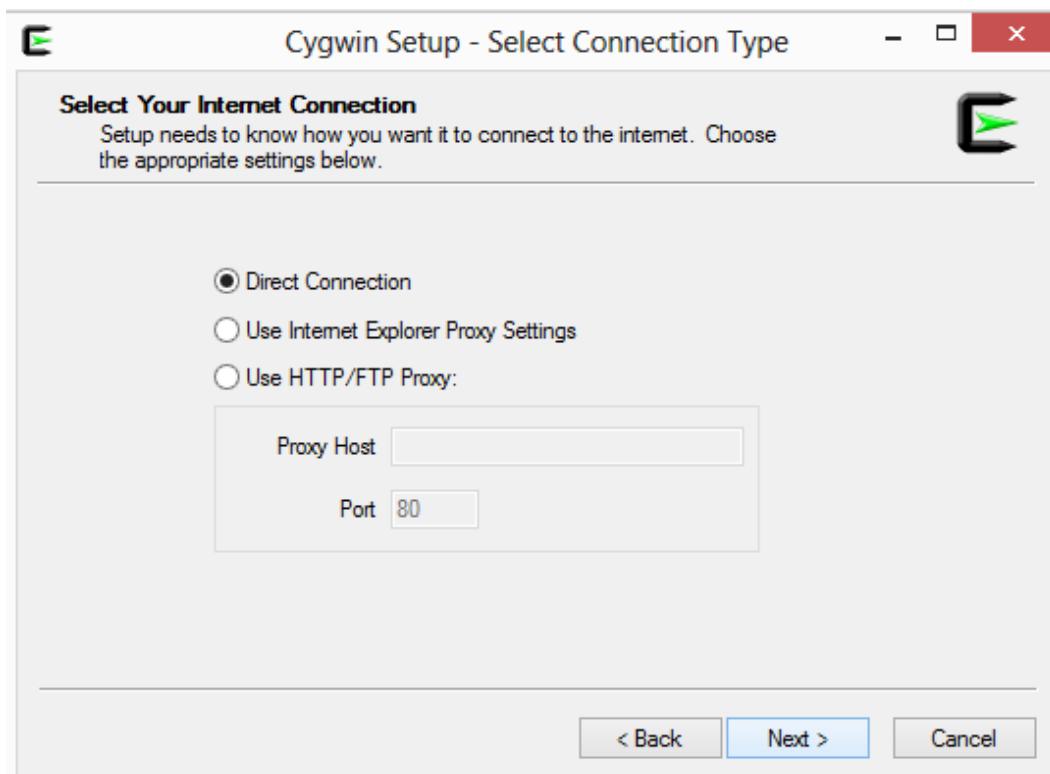


Figure 16: Select “Direct Connection”.

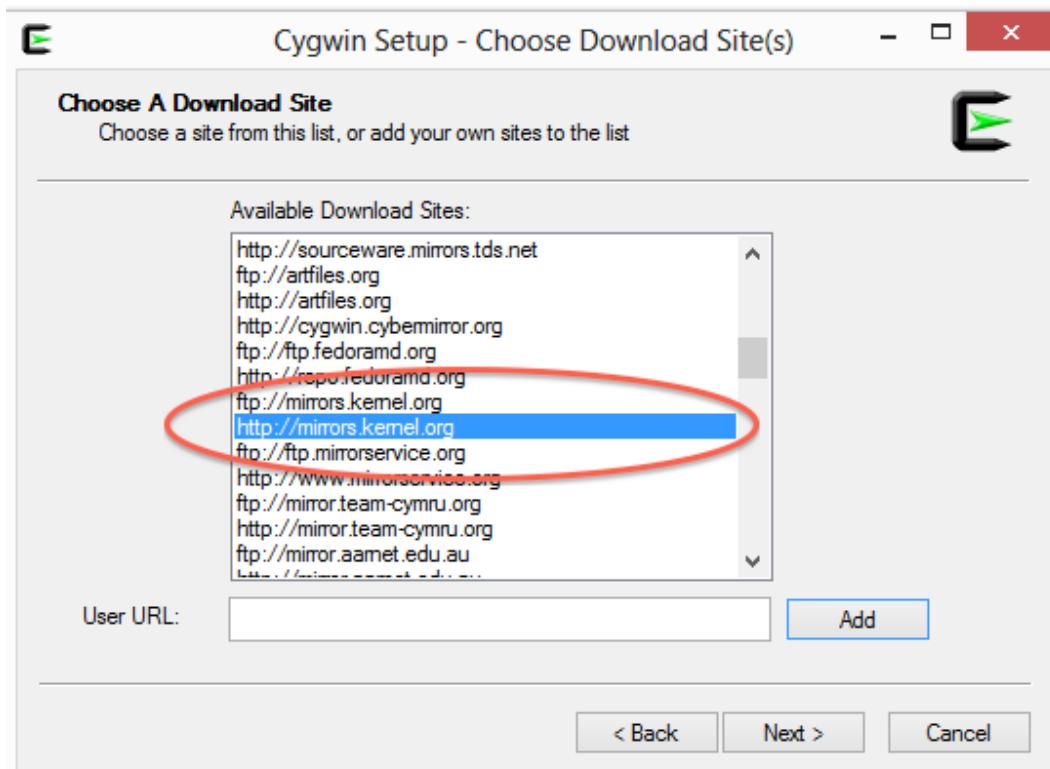


Figure 17: Use `http://mirrors.kernel.org` for fast downloading. This is the Linux Kernel site.

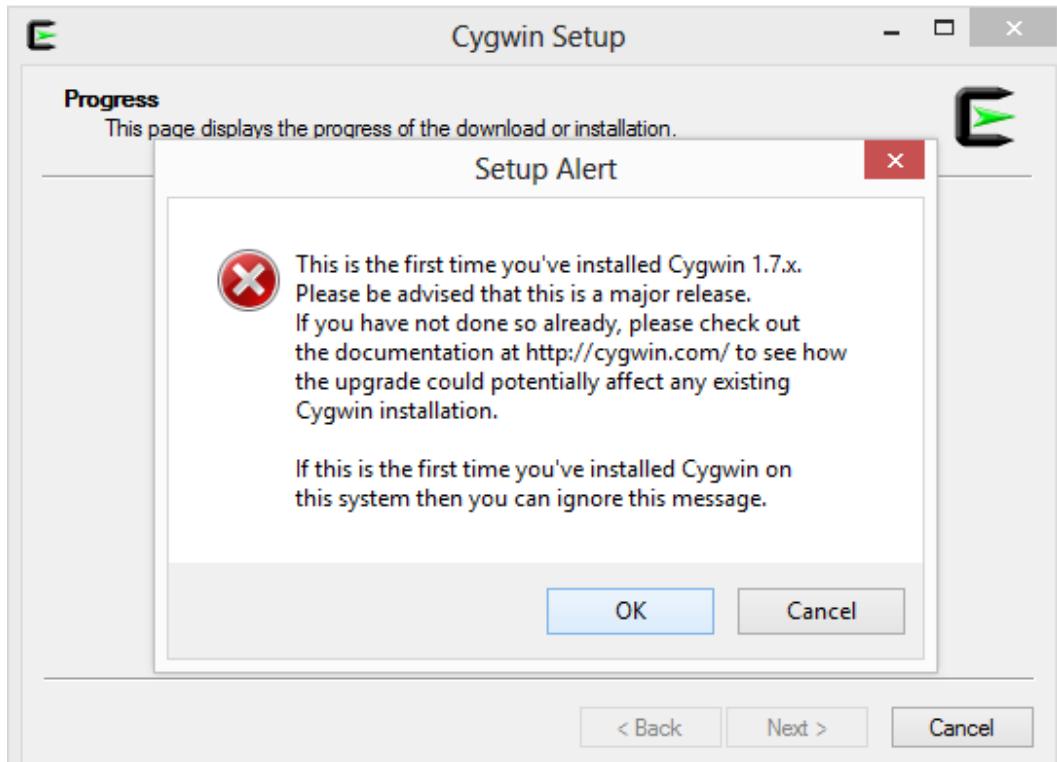


Figure 18: Click *OK* here.

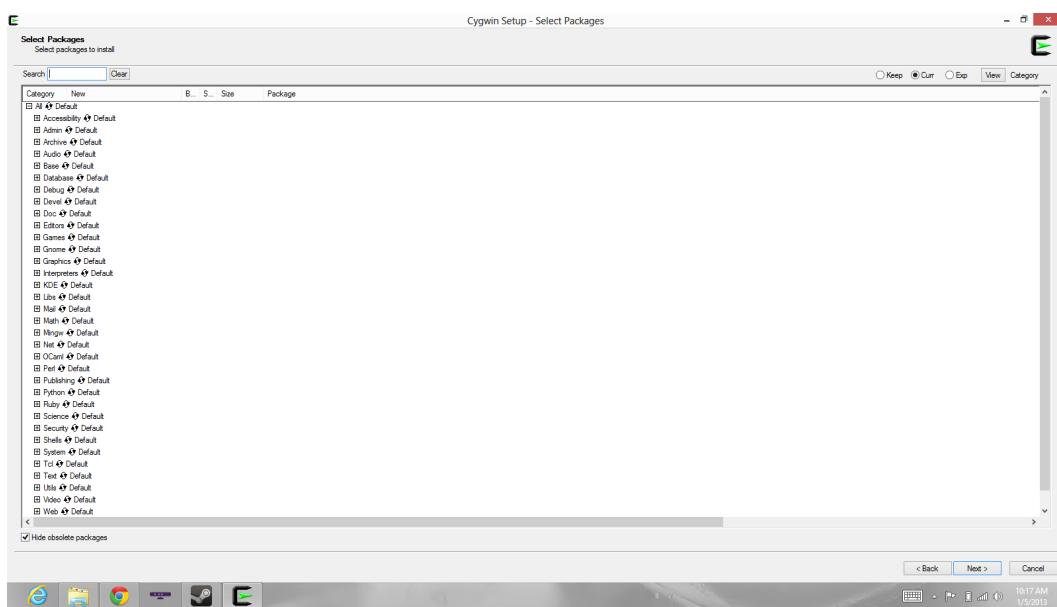


Figure 19: You now need to select some packages to install.

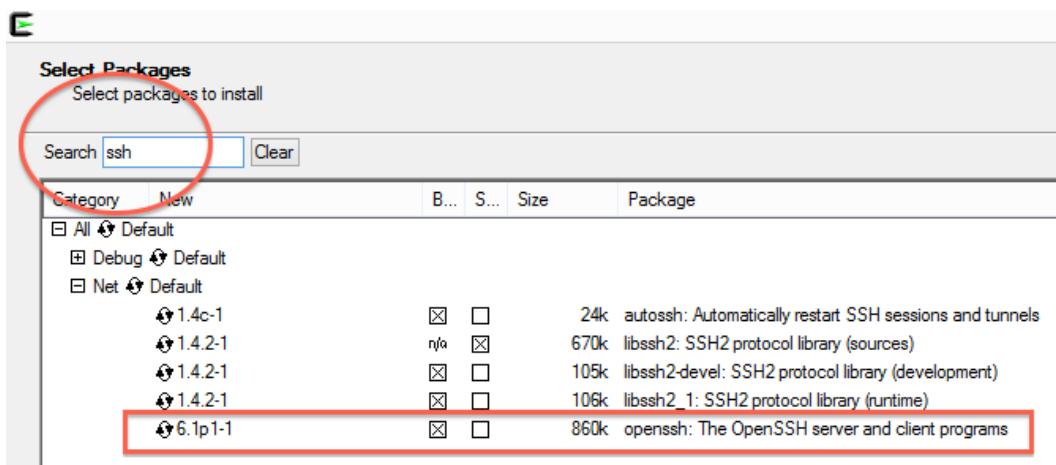


Figure 20: Type in *ssh* and select OpenSSH (what's shown is overkill but doesn't hurt).

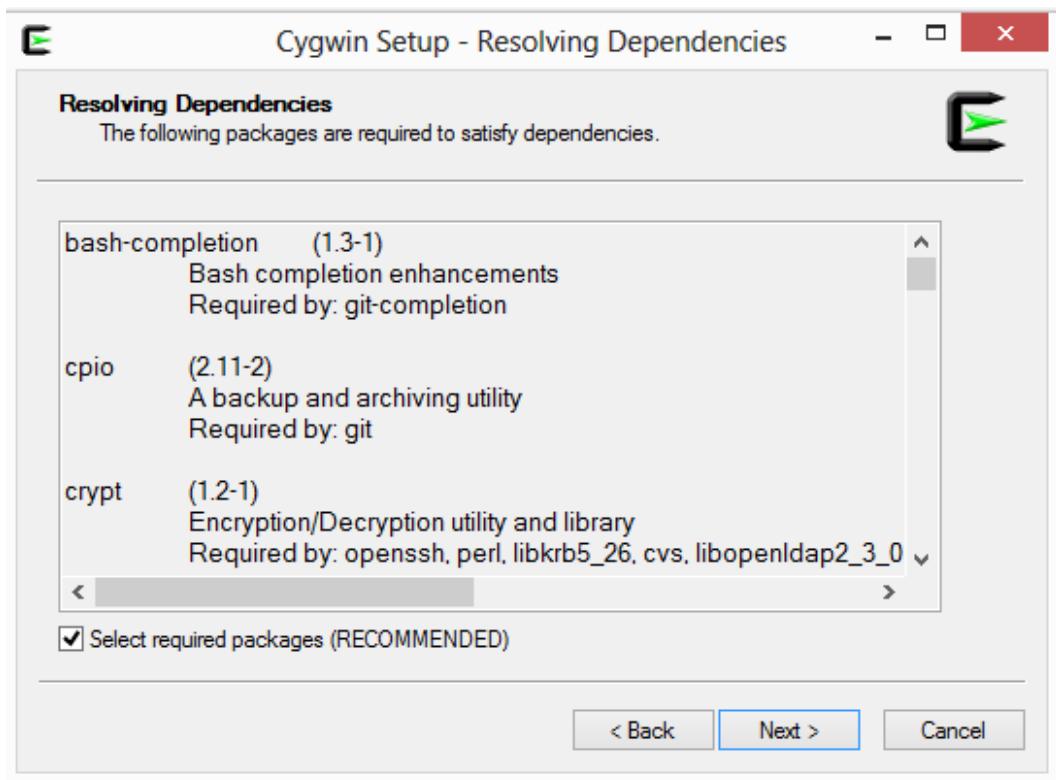


Figure 21: Go ahead to the next screen.

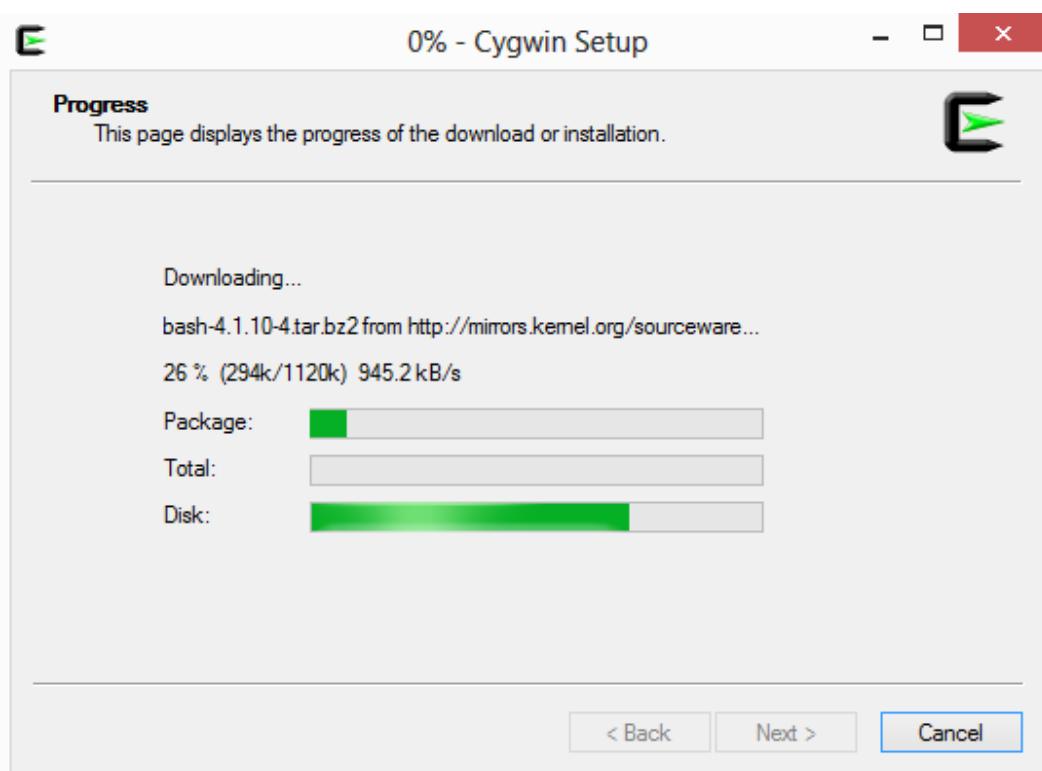


Figure 22: Wait for setup to complete.

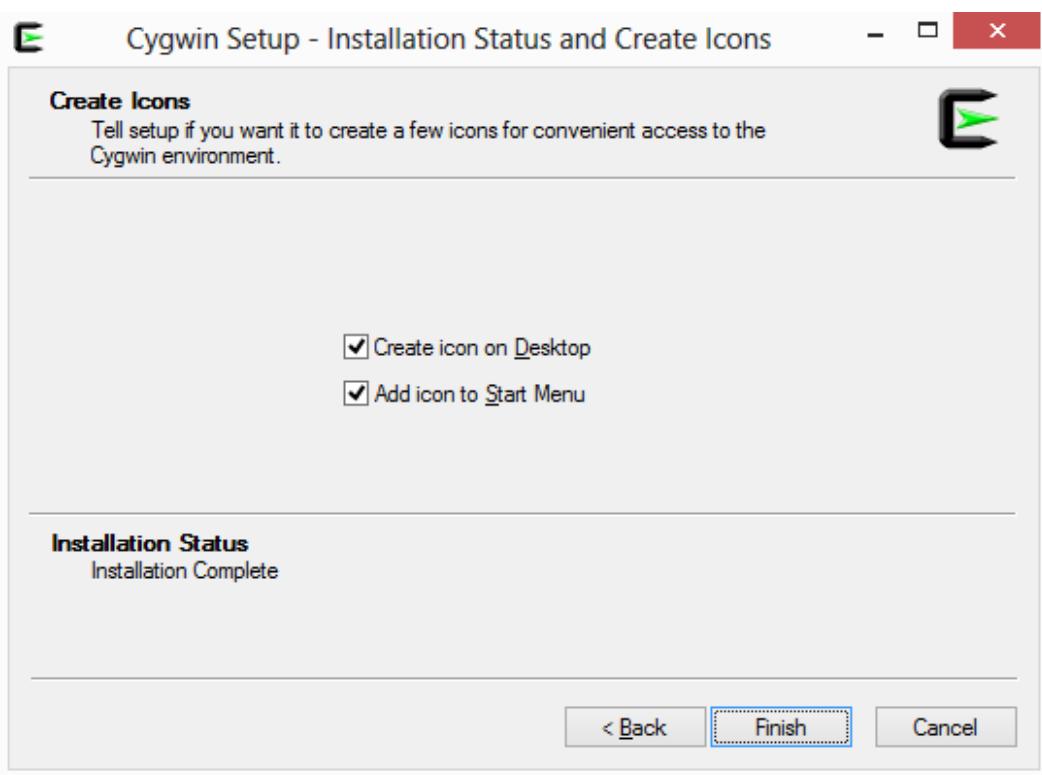


Figure 23: You now have a command-line interface on Windows. Double click the Desktop Cygwin icon to boot up a terminal.

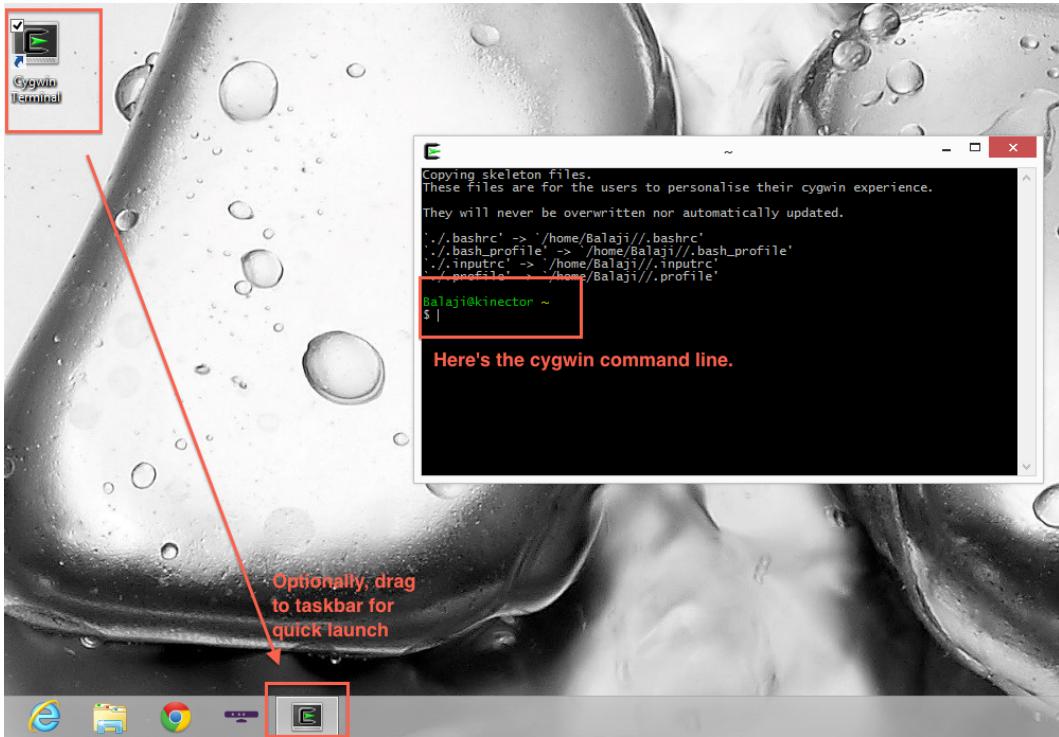


Figure 24: Upon first boot, your Cygwin terminal should show a message like this.

```
Balaji@kinector ~  
$ ping google.com  
Pinging google.com [74.125.227.105] with 32 bytes of data:  
Reply from 74.125.227.105: bytes=32 time=65ms TTL=51  
Reply from 74.125.227.105: bytes=32 time=65ms TTL=51  
Balaji@kinector ~  
$
```

Figure 25: You can confirm it works by typing in `ping google.com` and seeing a Reply.

Besides Cygwin, it is also possible to use a pure SSH client like [Putty](#), a browser-based client like the [Chrome Secure Shell](#), or the native [Windows Powershell](#). The reason we prefer Cygwin is threefold. First, some commands are best done locally. Second, of these alternatives, the environment that Cygwin provides is closest to the environment on a real Unix box, so we don't need to change the instructions around too much for each platform. Third, the local Cygwin environment can be configured to be very similar to the remote environment on your AWS machine (though a Mac will get even closer).

- Linux: Terminal

If you are using Linux, you probably already know how to open up the Terminal. But just in case, here are [instructions for Ubuntu](#).

Sign up for AWS, Gravatar, Github, and Heroku

Now that you have your browser (Chrome) and your terminal (Terminal.app or Cygwin), you will begin by signing up for four webservices:

- Github: github.com
- Gravatar: gravatar.com
- Amazon Web Services: aws.amazon.com
- Heroku: heroku.com

Over the course of this class, you will use Github to store code, Gravatar to identify yourself in public git commits and pull requests, AWS as a development environment for writing code, and Heroku for deploying your code on the web.

- AWS Signup

Let's begin by getting you set up with Amazon Webservices. You will need an email account, your cellphone, and your credit card with you to begin. *Please use the same email account/phone/credit card for all services* used in this course; this will save you headaches related to service synchronization. Note also that the credit card will only be billed if you exceed the free usage tier. We'll talk about this later, and Amazon has some checks in place to prevent you from using too much, but right now you won't be billed. Begin by going to <http://aws.amazon.com> and then following the sequence of screens as shown below:

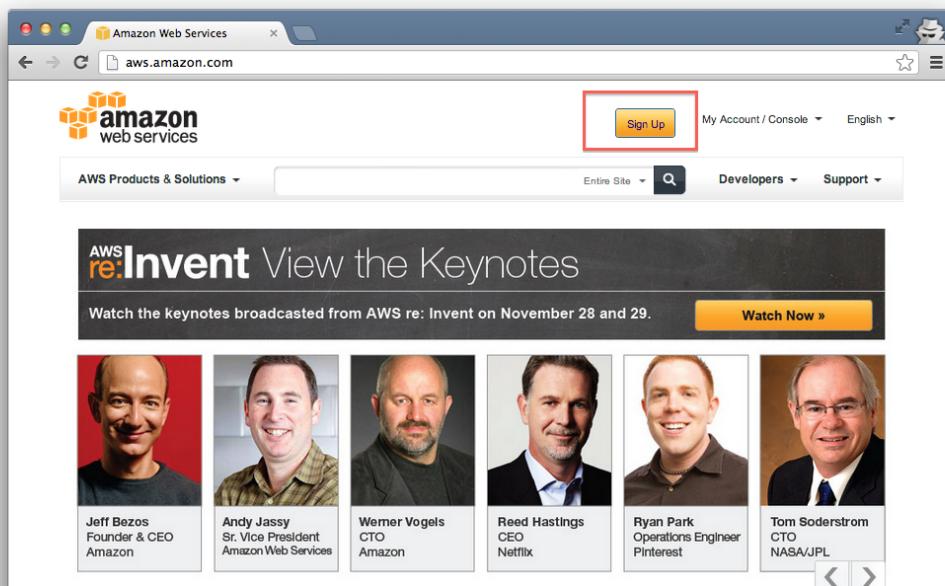


Figure 26: The AWS homepage. Click “Sign Up” at the top.

A screenshot of the AWS sign-up form. The page title is "Amazon Web Services Sign In or Create an AWS Account". It instructs users they can sign in with an existing Amazon.com account or create a new account by selecting "I am a new user". A radio button for "I am a new user" is selected. Below it, another radio button for "I am a returning user and my password is:" is present, followed by a text input field. A "Sign in using our secure server" button is shown with a small arrow icon. Below the button are links for "Forgot your password?" and "Has your e-mail address changed?". At the bottom, there is a link to "Learn more about AWS Identity and Access".

Figure 27: The signup form.

The screenshot shows a web browser window for the Amazon Web Services (AWS) sign-in page. The URL in the address bar is https://www.amazon.com/ap/signin?openid.assoc_handle=aws&openid.return_to=https%3A%2F%2Fportal.aws.amazon..... The page features the Amazon Web Services logo at the top left. Below it, the heading "Sign In or Create an AWS Account" is displayed in orange. A sub-instruction below the heading reads: "You may sign in using your existing Amazon.com account or you can create a new account by selecting "I am a new user.\"" A text input field contains the email address "startup.stanford.edu@gmail.com". Below the input field are two radio button options: "I am a new user." (which is selected and highlighted with a red border) and "I am a returning user and my password is:". There is also a link "Sign in using our secure server" with a magnifying glass icon. At the bottom of the form, there are links for "Forgot your password?", "Has your e-mail address changed?", and "Learn more about AWS Identity and Access".

Figure 28: Register as a new user.

The screenshot shows a web browser window for the AWS login credentials setup page. The URL in the address bar is <https://www.amazon.com/ap/register?ie=UTF8&openid.ns.pape=http%3A%2F%2Fspecs.openid.net%2Fextensions%2Fp...>. The page features the Amazon Web Services logo at the top left. The heading "Login Credentials" is displayed in orange. A sub-instruction below the heading reads: "Use the form below to create login credentials that can be used for AWS as well as Amazon.com." The form includes fields for "My name is:" (Startup Stanford), "My e-mail address is:" (startup.stanford.edu@gmail.com), and "Type it again:" (startup.stanford.edu@gmail.com). Below these fields is a note: "note: this is the e-mail address that we will use to contact you about your account". There are also fields for "Enter a new password:" and "Type it again:", both containing masked text. A "Continue" button with a magnifying glass icon is located at the bottom right. At the very bottom of the page, there is a link "About Amazon.com Sign In".

Figure 29: Set up your login.

Amazon Web Services Sign Up

Contact Information

* required fields

Full Name*: Startup Stanford

Company Name: Stanford University

Country*: United States

Address Line 1*: [REDACTED]
Street address, P.O. box, company name, c/o

Address Line 2: [REDACTED]
Apartment, suite, unit, building, floor, etc.

City*: [REDACTED]

State, Province or Region*: [REDACTED]

ZIP or Postal Code*: [REDACTED]

Phone number*: [REDACTED]

Security Check

Image:

LHRPRT

Try a different image

Why do we ask you to type these characters?

Type the characters in the above image*: LHRPRT

Having Trouble? Contact us.

AWS Customer Agreement

Check here to indicate that you have read and agree to the terms of the Amazon Web Services Customer Agreement. [\[link\]](#)

Create Account and Continue 

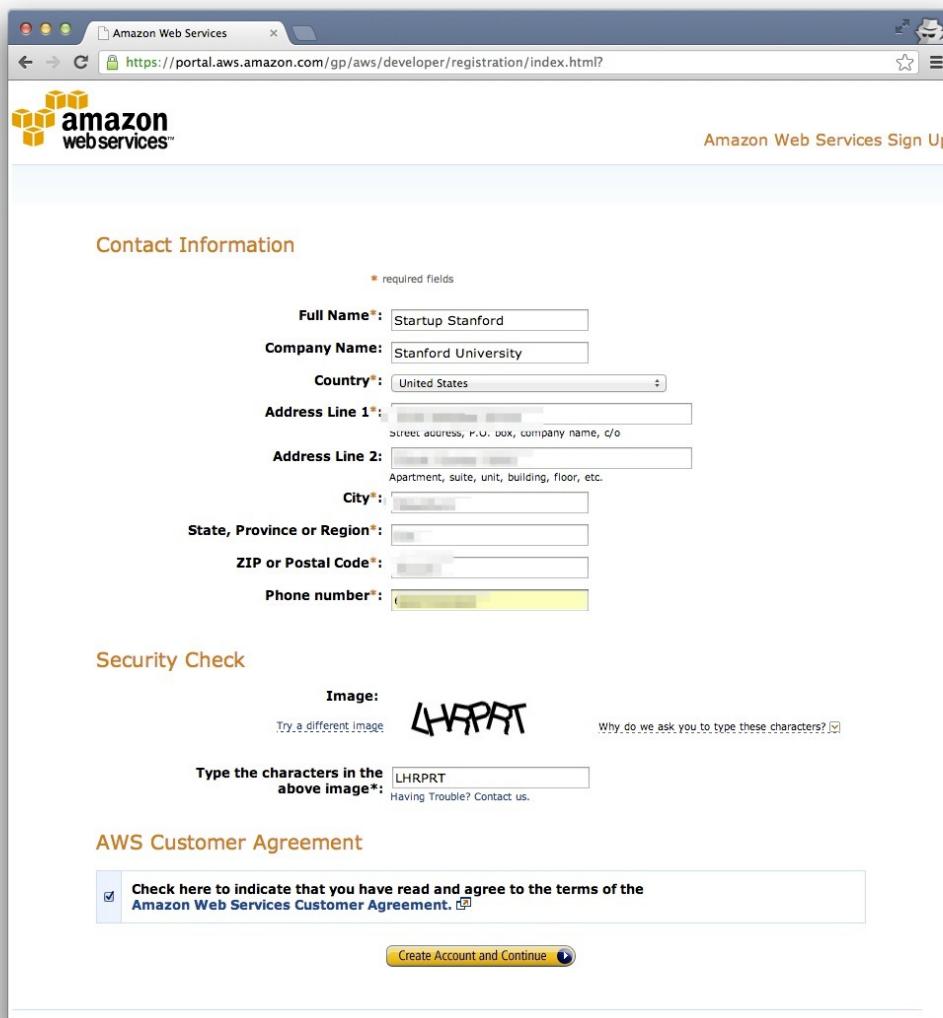


Figure 30: Enter address and complete CAPTCHA.

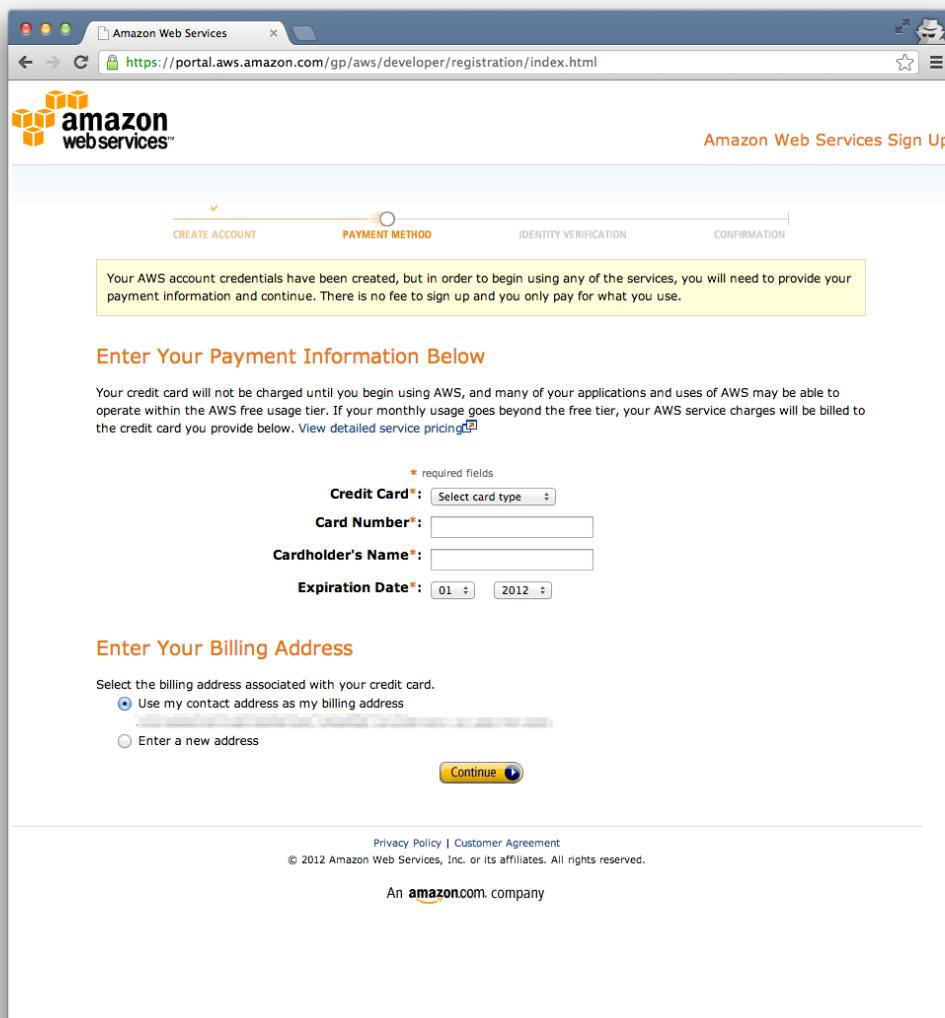


Figure 31: Set up your credit card.

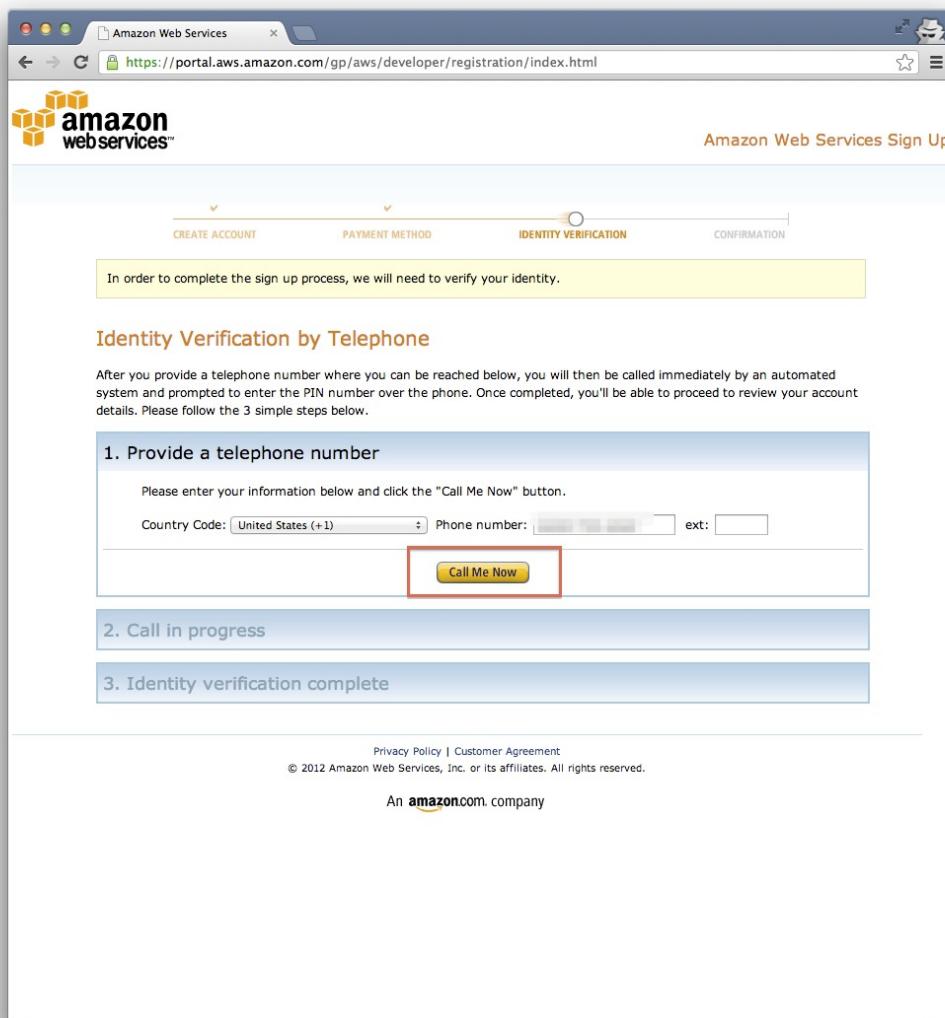


Figure 32: Begin phone verification.

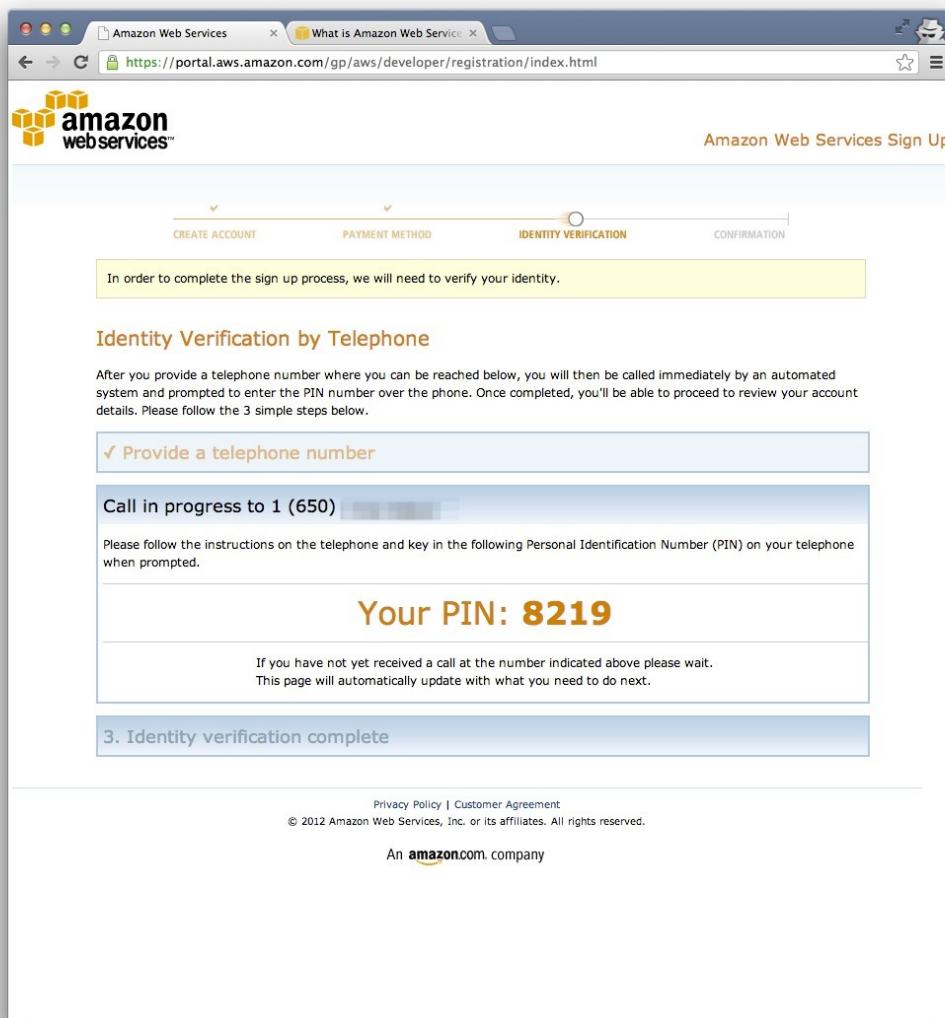


Figure 33: When called, enter in your PIN followed by an asterisk ('*').

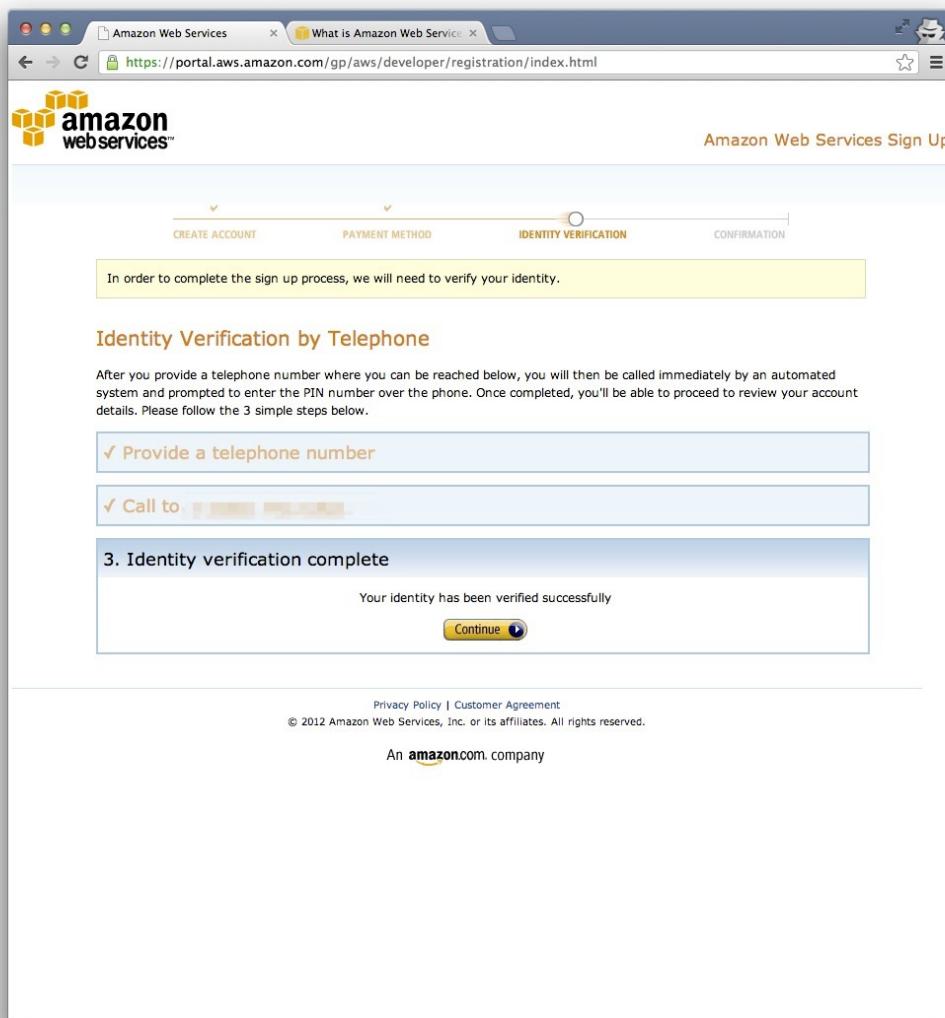


Figure 34: *Identity verification complete.*

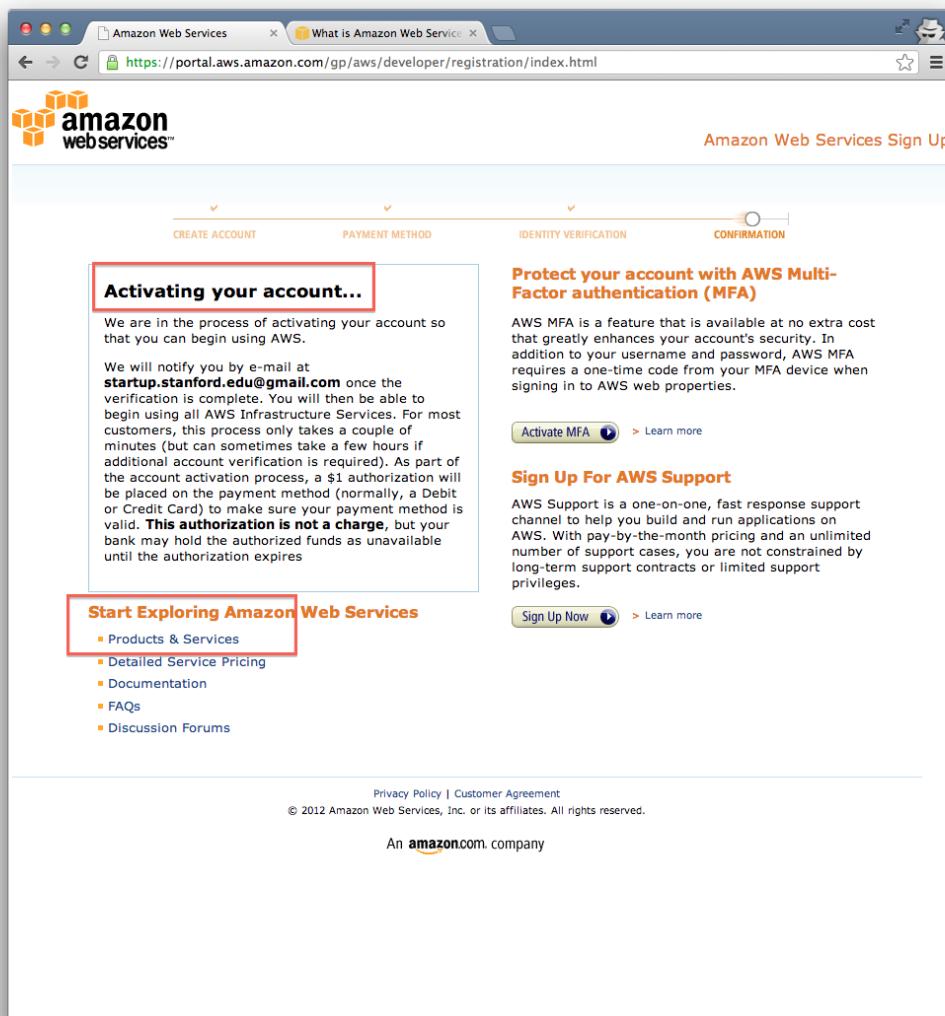


Figure 35: Account activation is now beginning. This may take a while.

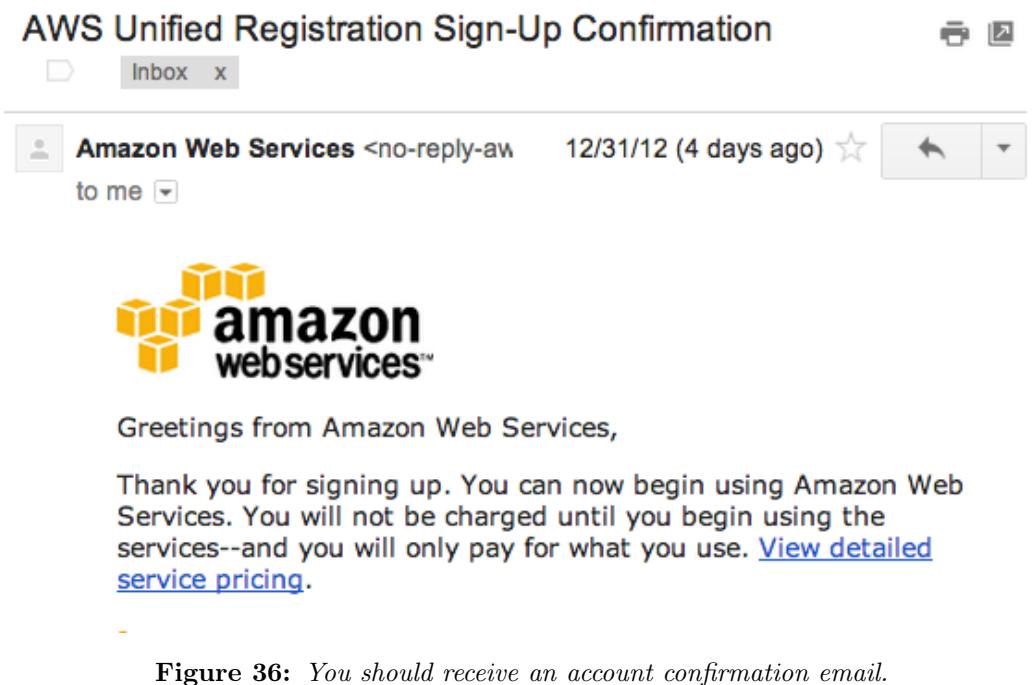


Figure 36: You should receive an account confirmation email.

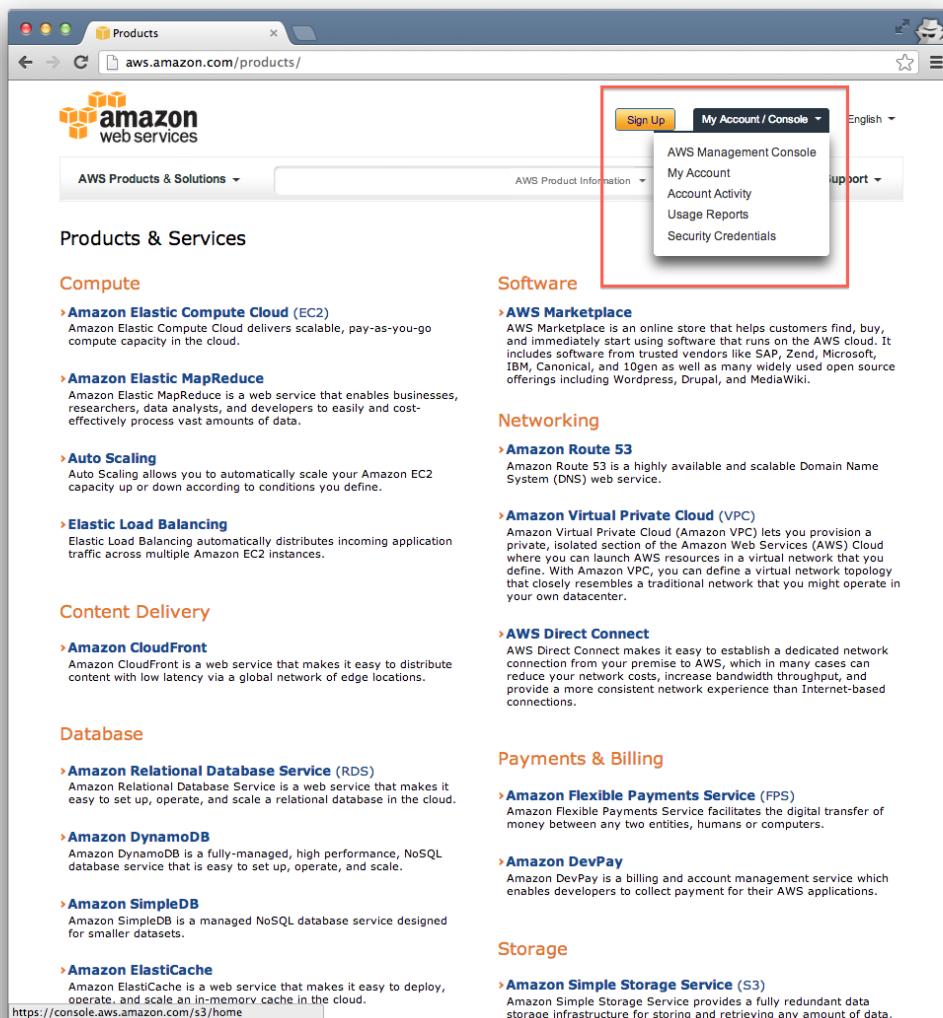


Figure 37: Go to aws.amazon.com/products and click 'My Account Console' to log in.

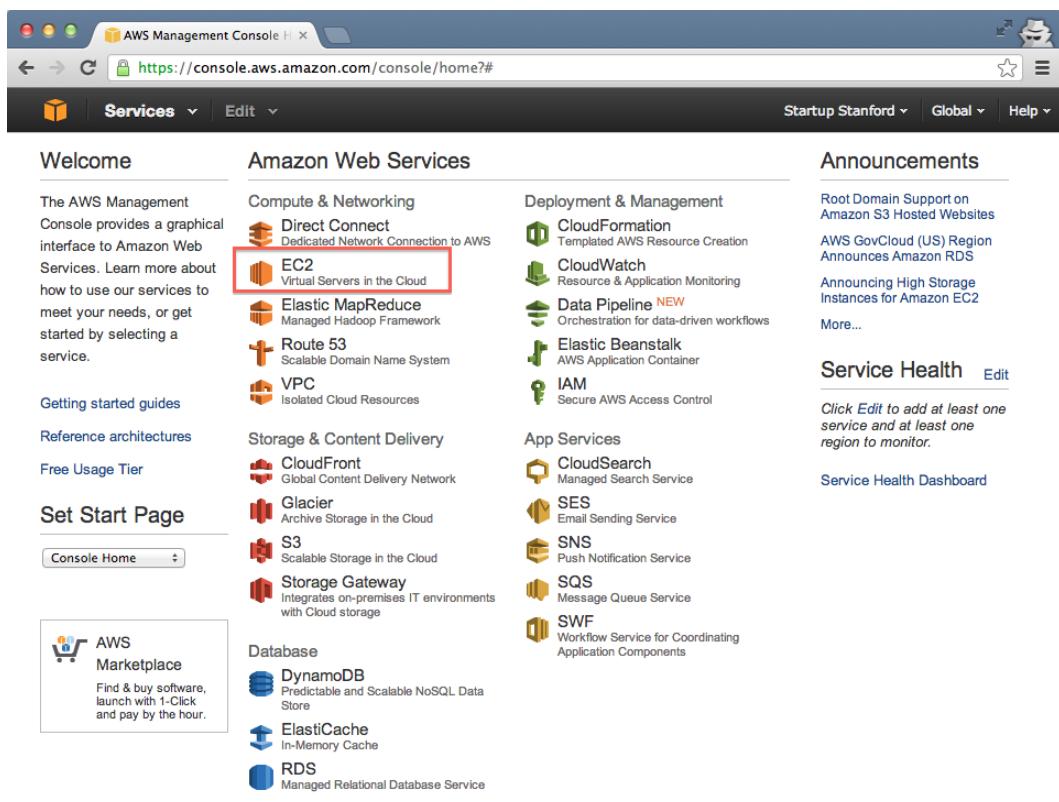


Figure 38: Now you have an active AWS dashboard. EC2 (boxed) will be of particular interest to us soon.

Fantastic. You now have access to a variety of AWS services. Let's keep going and we'll come back to this dashboard as the class progresses.

- Gravatar Signup

The next step is to get your Gravatar account set up, so that you have a *globally recognizable avatar* that will be used to identify your git commits. A Gravatar is very similar to the small image that appears next to your comments on Facebook, except that it is used by sites outside of Facebook, such as Github. Begin by going to <http://www.gravatar.com> and then work through the following steps:

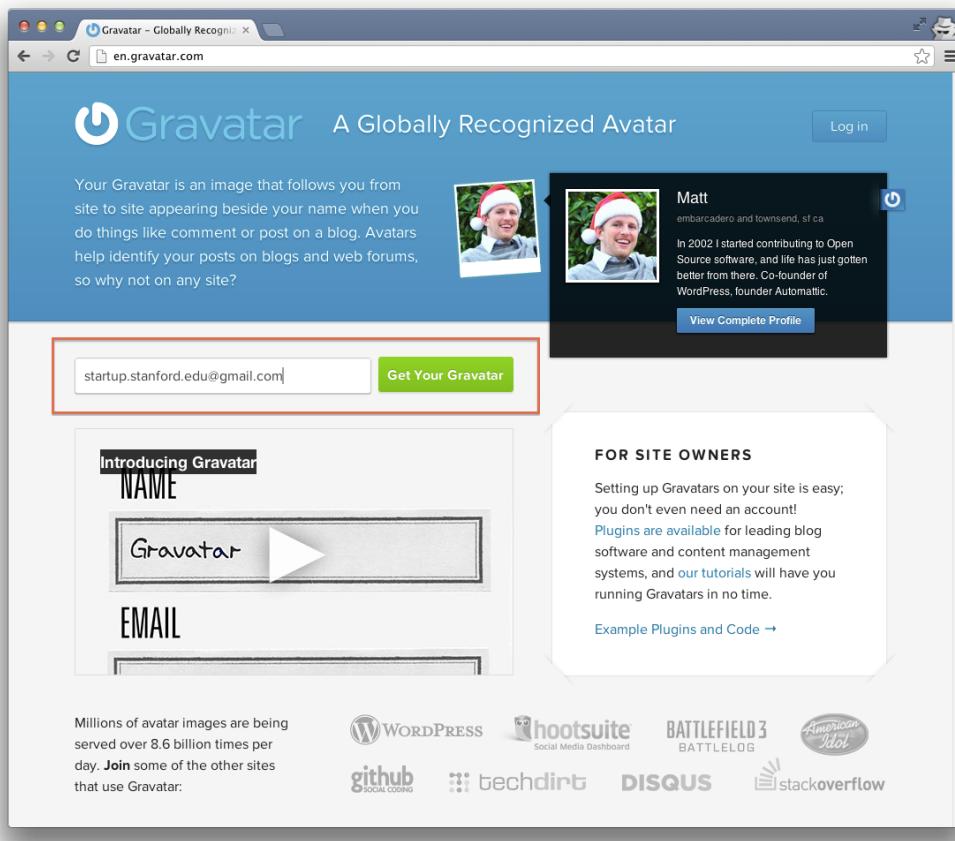


Figure 39: Go to the gravatar homepage to get started.

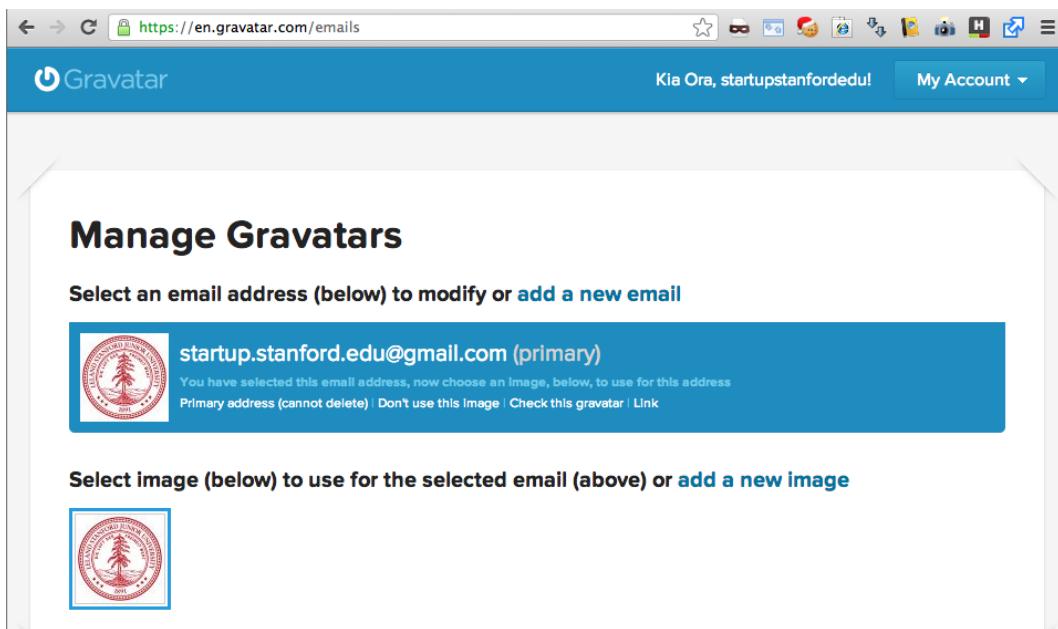


Figure 40: Once everything is set up, your page should look like this.

The Gravatar site has been buggy in the past and you may need to repeat some of the steps (particularly the cropping step and the “add a new image” step at the very end) a few times. Do this by logging out of Gravatar, clearing cookies in Chrome, logging back in and resuming where you left off.

- Github Signup

Github is one of the most important developments in software over the past five years. The site began in 2008 as a simple frontend for the open-source `git` distributed version control tool, which we will cover in much more detail later in the course. In this sense it was similar to code repositories like SourceForge, Google Code, or Bitbucket. But Github has transcended the category to become an institution in its own right. Features that it pioneered (like web-based pull requests and `README.md` files) are now crucial tools for any group software engineering work. Moreover, `github.com` is now the world’s largest and most dynamic repository of open-source code, and a strong Github account has become **more important** than your CV or formal education for the most cutting-edge technology companies. Let’s get started by navigating to <http://www.github.com>.

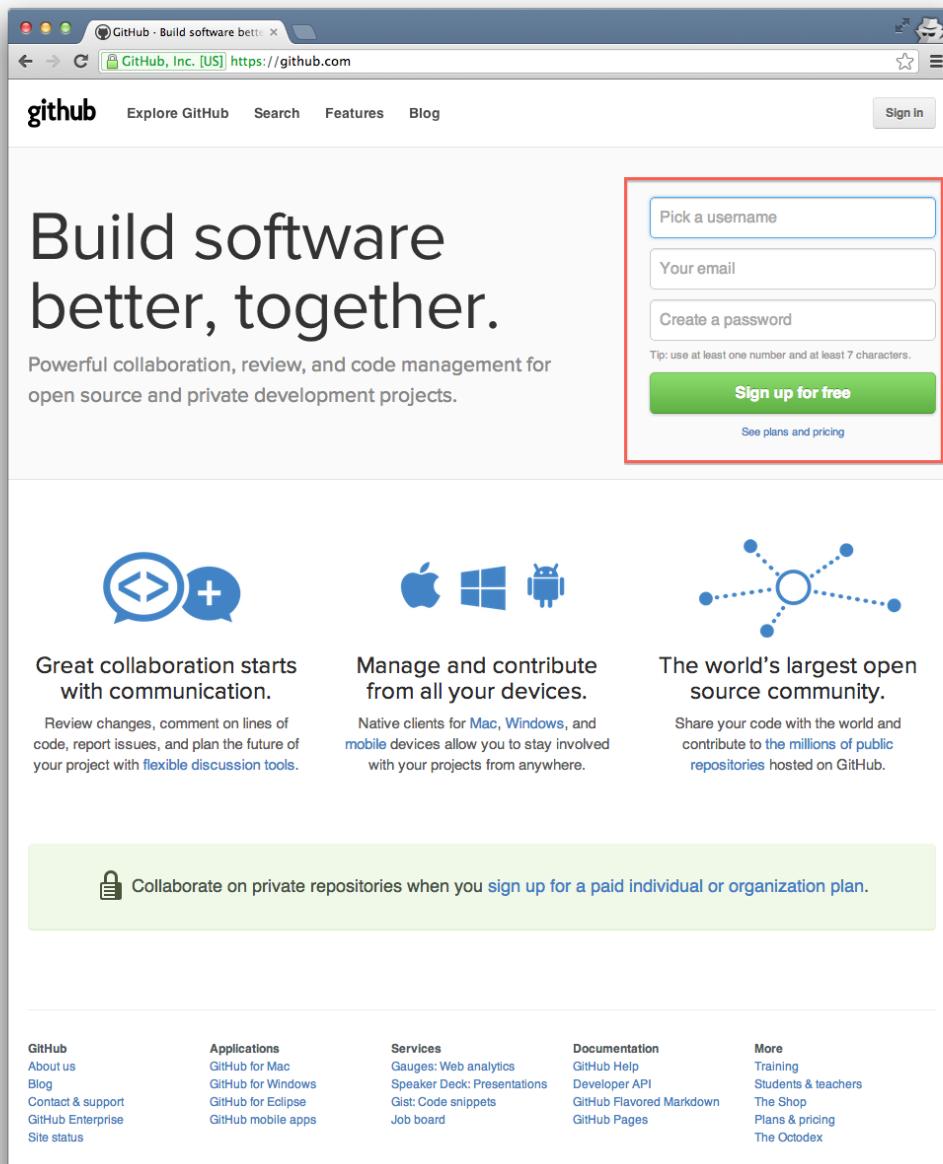


Figure 41: Sign up in the top right and confirm your account via email.

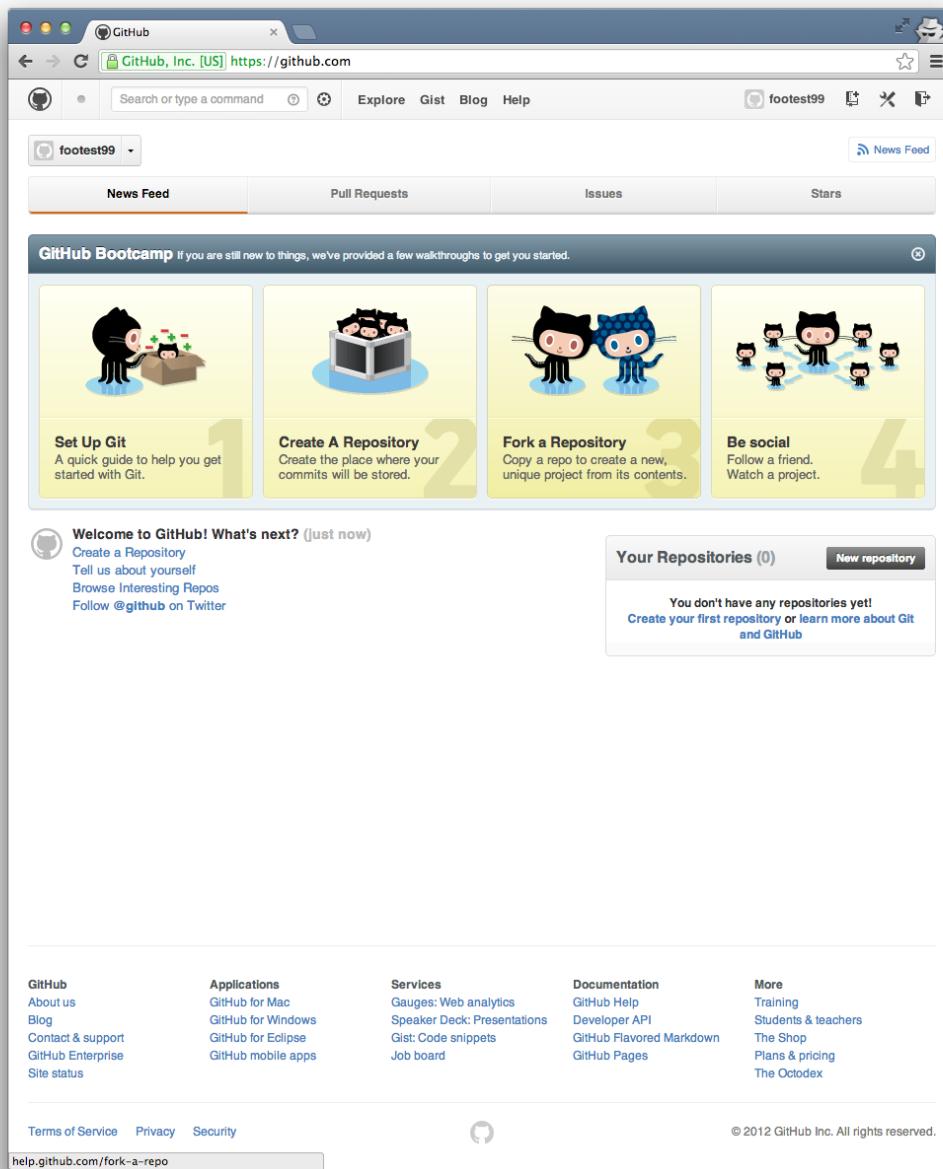


Figure 42: After signing in, you will see this page.

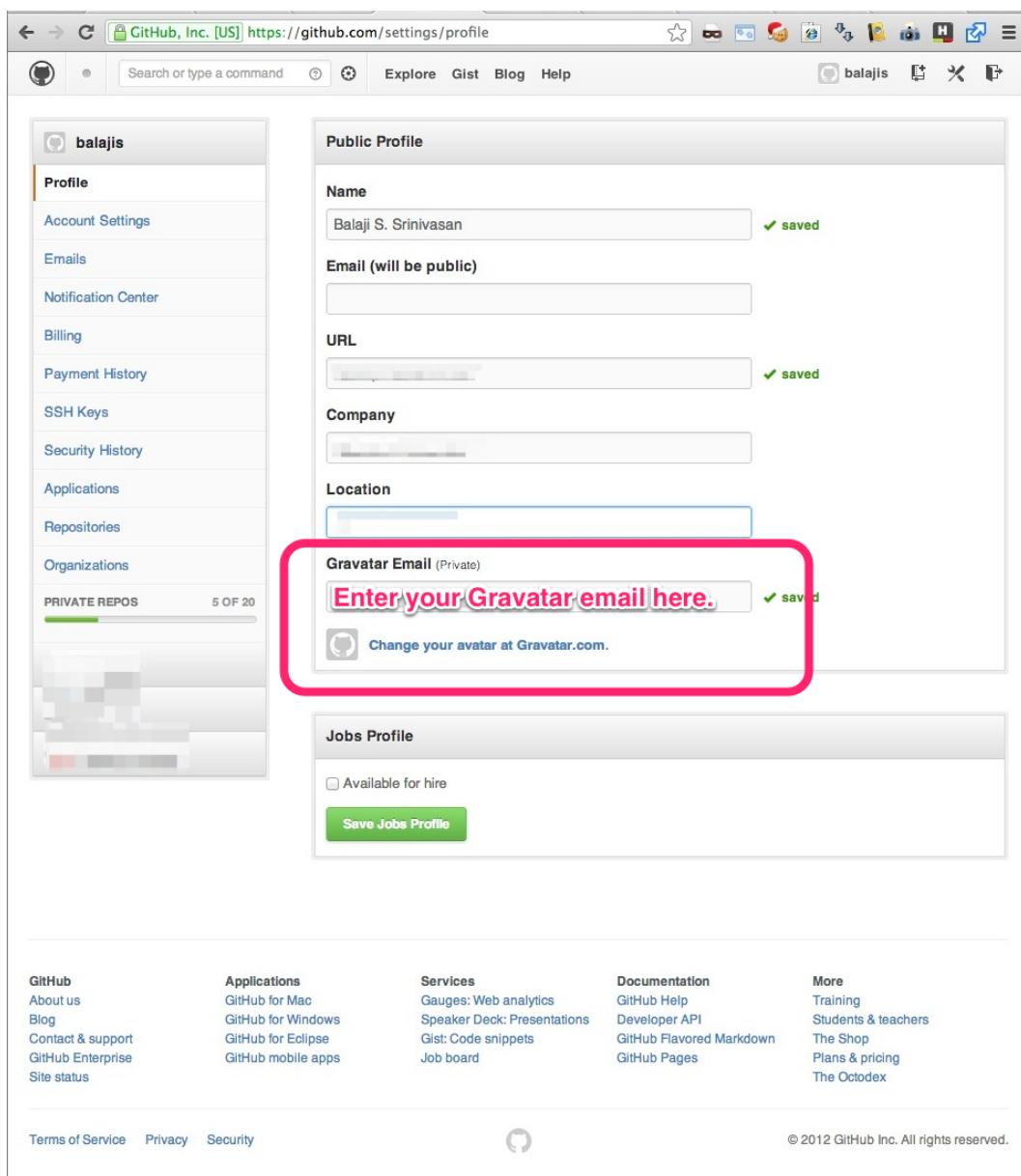


Figure 43: Navigate to `github.com/settings/profile` to enter your Gravatar email

You now have a Github account, which you will use for hosting your sourcecode for this class and likely for future projects. Take a look at your [profile page](#) in particular.

Importantly, if you [fork](#) a repository, you can optionally submit a so-called *pull request* to the owner of the upstream repository to merge in your contributions. As an analogy, this is like making a copy of a book (forking), writing up a new chapter (editing), and then sending that chapter back to the original author to integrate into the main book should he deem it worthy (issuing a pull request, i.e. a request to pull your code back into the main book). We'll cover these concepts in more depth later in the class.

- Heroku Signup

The last account we want to get configured is on Heroku, which began as a layer on top of Amazon Web Services. While AWS focuses more on low-level hardware manipulations, Heroku makes it easy to deploy a web/mobile application (see [here for more](#)). Begin by going to <http://www.heroku.com>.

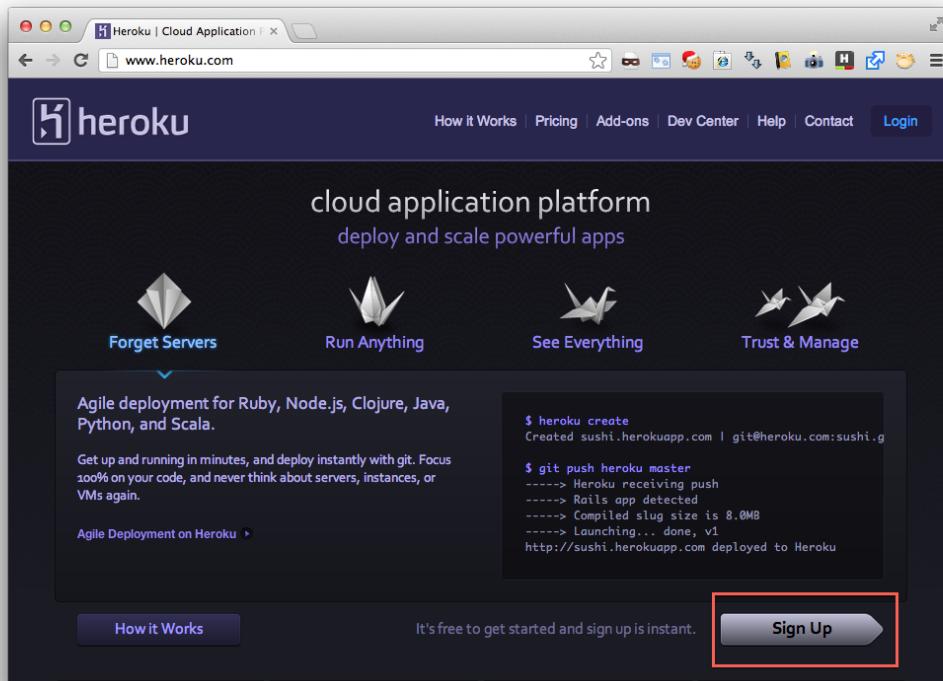


Figure 44: The Heroku homepage. Click “Sign Up” in the bottom right.

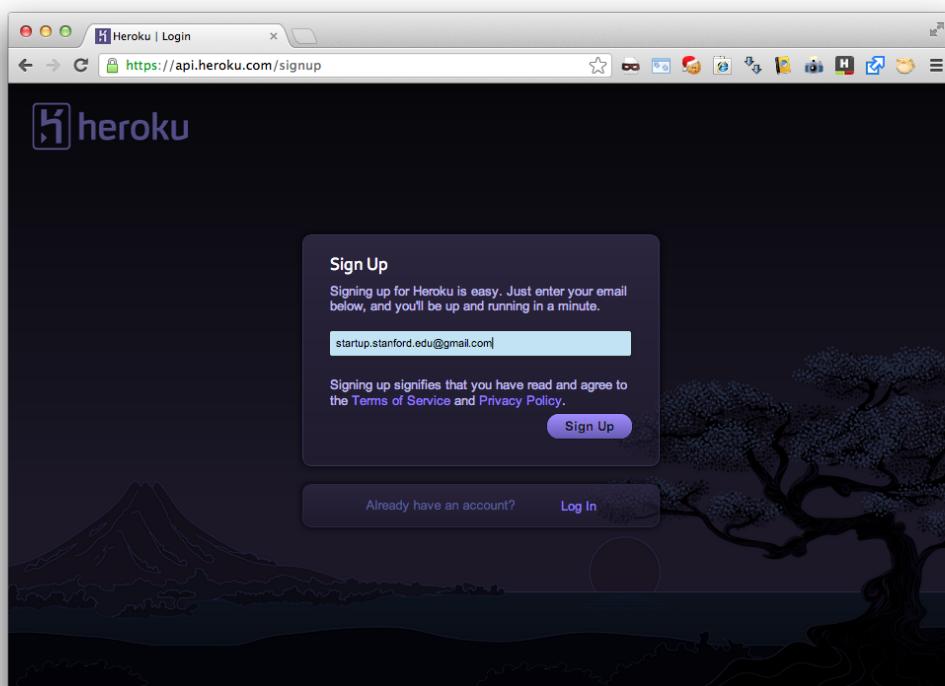


Figure 45: Enter your email.

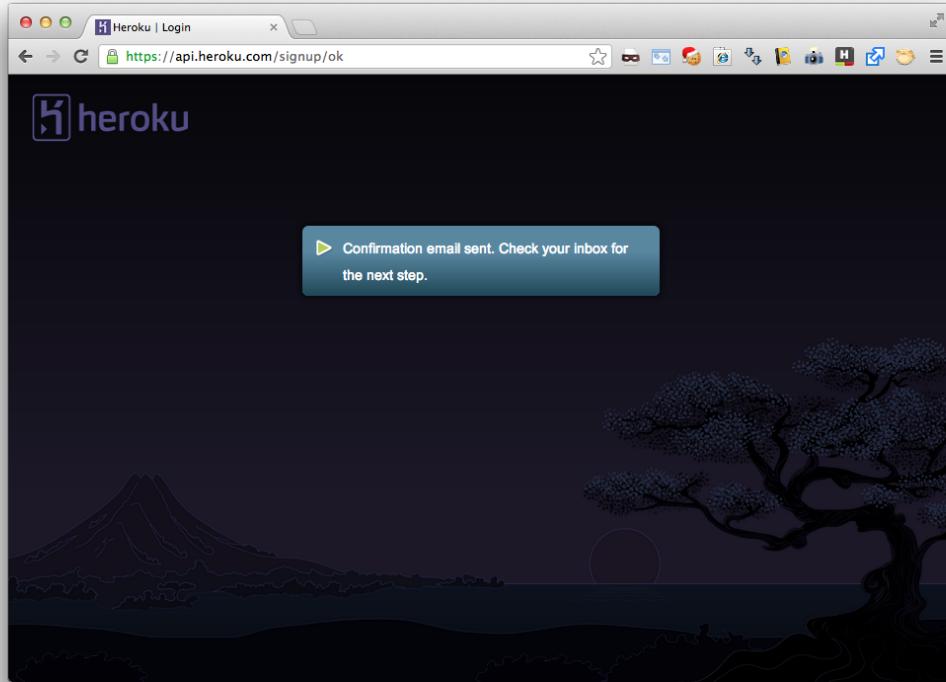


Figure 46: You should see a confirm page.

Confirm your account on Heroku

Inbox x

Heroku <bot@heroku.com> 4:05 PM (0 minutes ago)

to me

Thanks for signing up with Heroku! You must follow this link to activate your account:

<https://api.heroku.com/signup/accept2/1>

Have fun, and don't hesitate to contact us with your feedback.

- The Heroku Team
<http://heroku.com/>

Heroku is the cloud platform for rapid deployment and scaling of web applications.
Get up and running in minutes, then deploy instantly via Git.

To learn more about Heroku and all its features, check out the Dev Center:
<http://devcenter.heroku.com/articles/quickstart>

Figure 47: Check your email and click the confirm link.

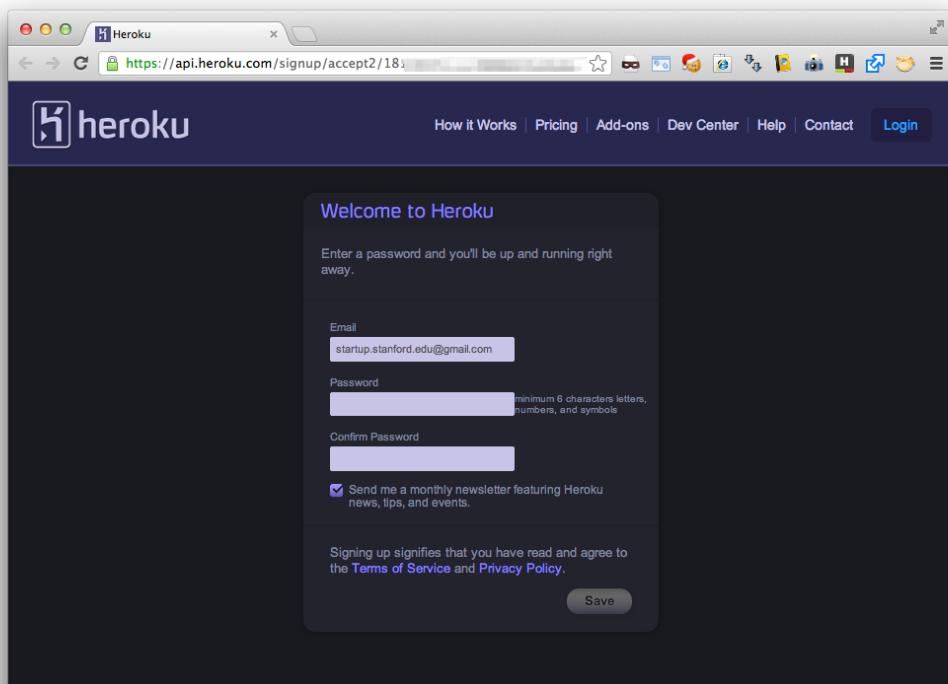


Figure 48: Now set up your password.

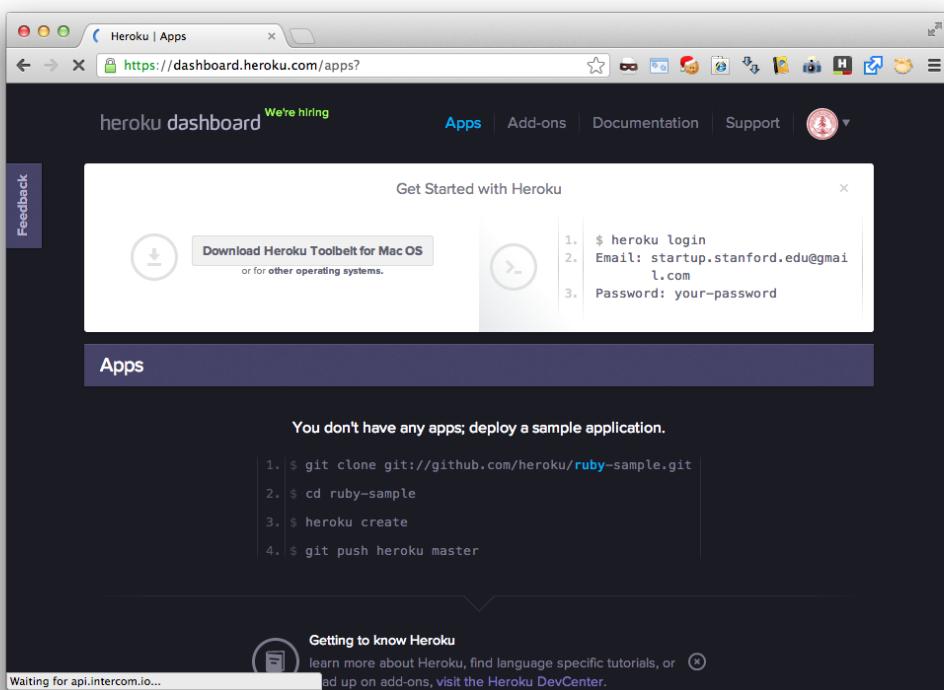


Figure 49: You now have a Heroku account. Ignore the instructions for now, we'll return to them later.

All right. At this point you should have your AWS, Gravatar, Github, and Heroku accounts all set up. Our next step will be to get an AWS instance running in the cloud as a development environment.

Connect to a Cloud Computer

Launch your EC2 Instance

You are now going to launch and connect to an AWS cloud computer; you may find [Amazon's own instructions](#) useful as a supplement. We are going to initialize Amazon's smallest kind of cloud computer (a `t1.micro` instance) running [Ubuntu 12.04.2 LTS](#) (a popular Linux distribution) in its `us-east-1` region. Begin by returning to <http://aws.amazon.com> and log into the dashboard as shown.

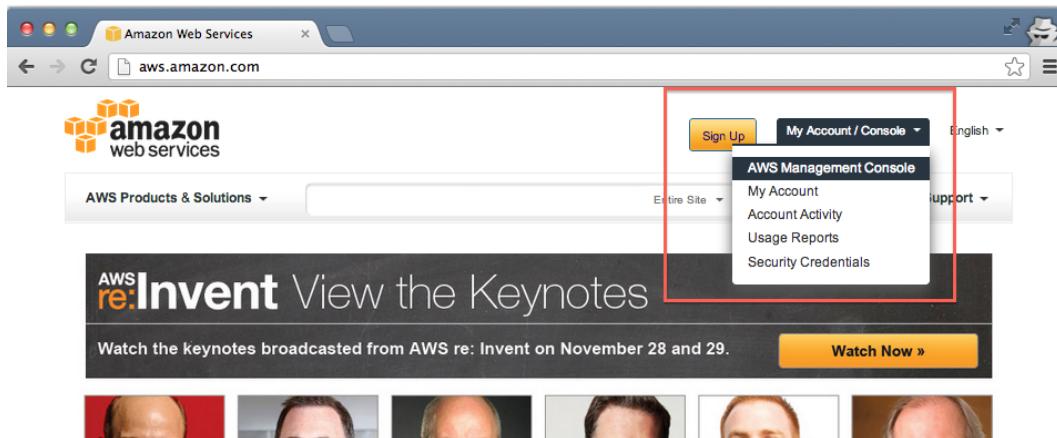


Figure 50: Log into your Management Console.

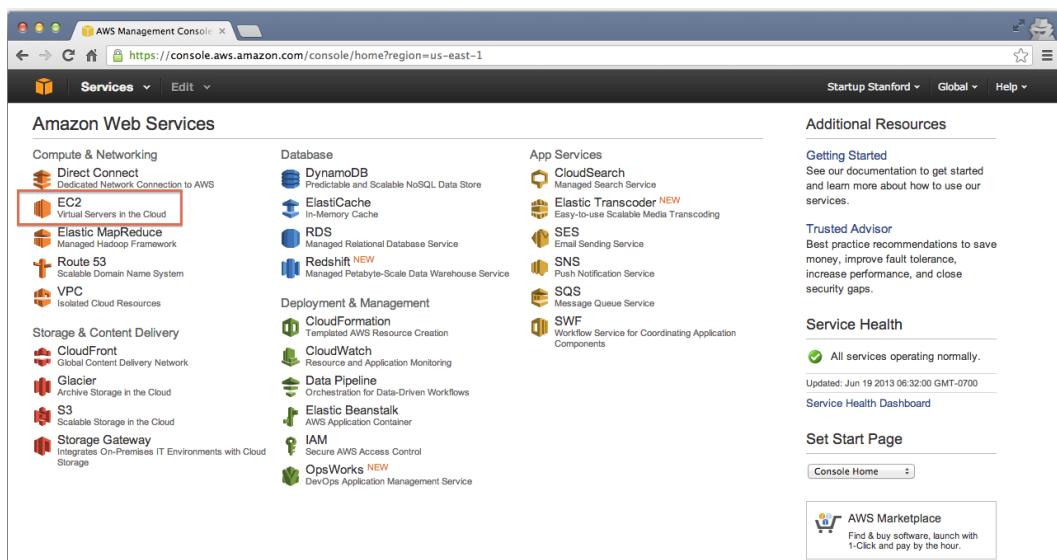


Figure 51: Click "EC2" on the AWS Dashboard.

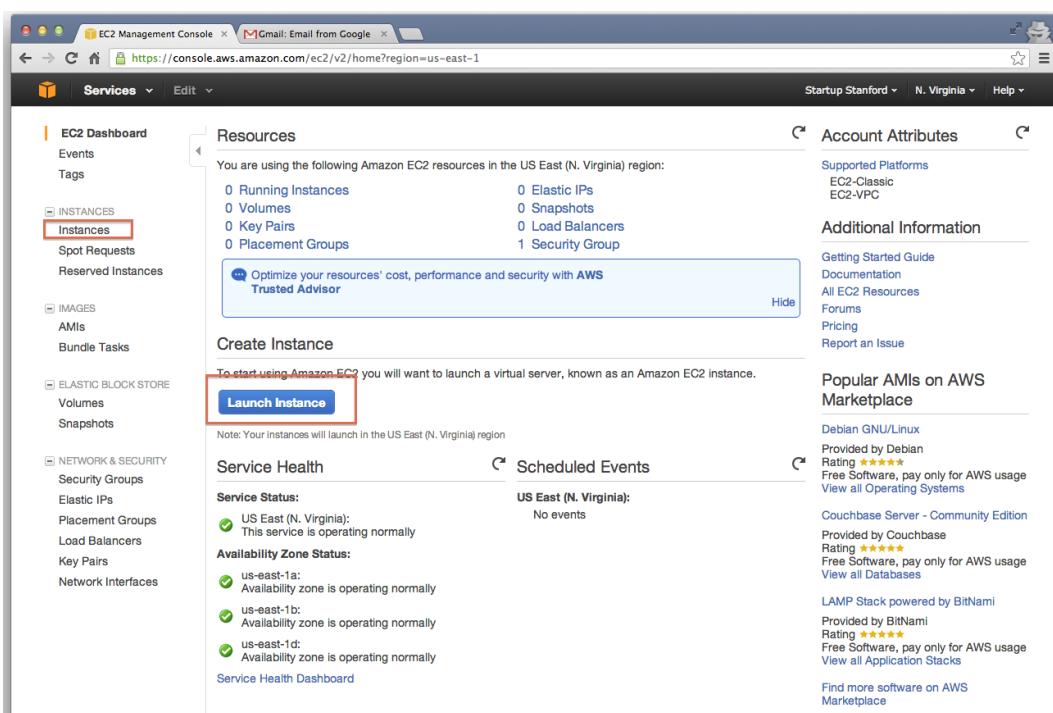


Figure 52: Click “Launch Instances”. Also note the “Instances” link (boxed) on the left hand side; you will need this in a bit.

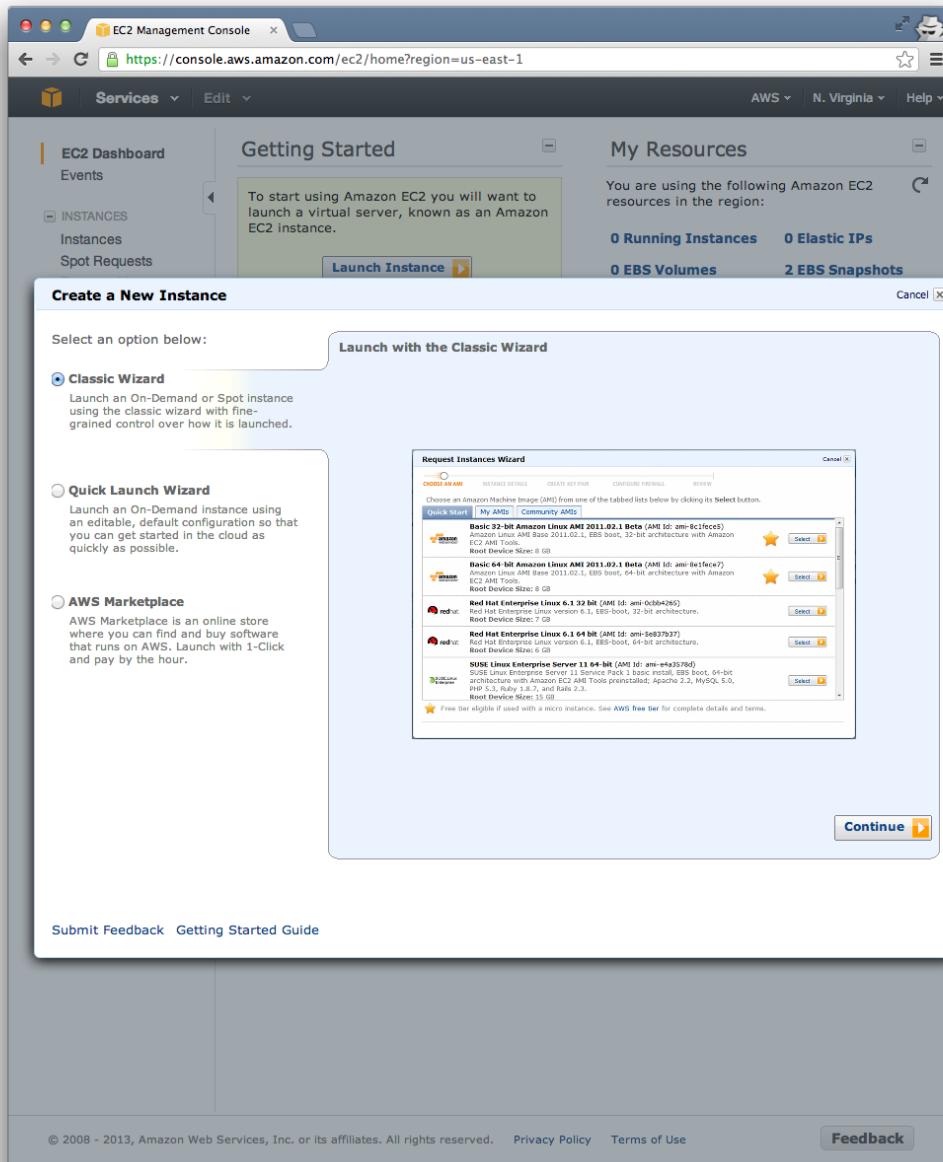


Figure 53: You should see the Create a New Instance Wizard.

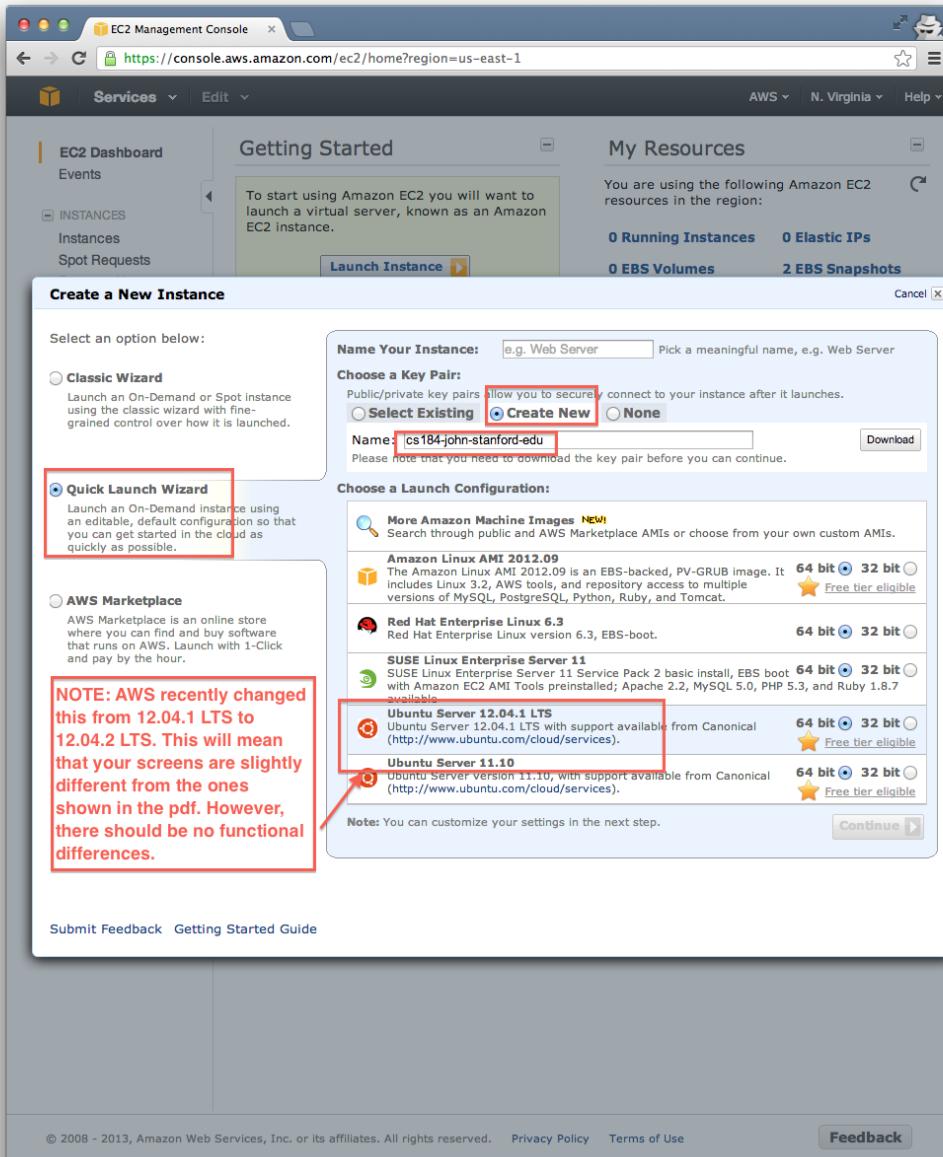


Figure 54: Select the *Quick Launch Wizard*, click the “*Create New*” radio button, enter a memorable keyname (e.g. *cs184-john-stanford-edu*), and select *Ubuntu Server 12.04.2 LTS* for the “*Launch Configuration*”. Note that the remaining screenshots depict *12.04.1 LTS*, while your screen will show *12.04.2 LTS*; this won’t change anything that you do, however.

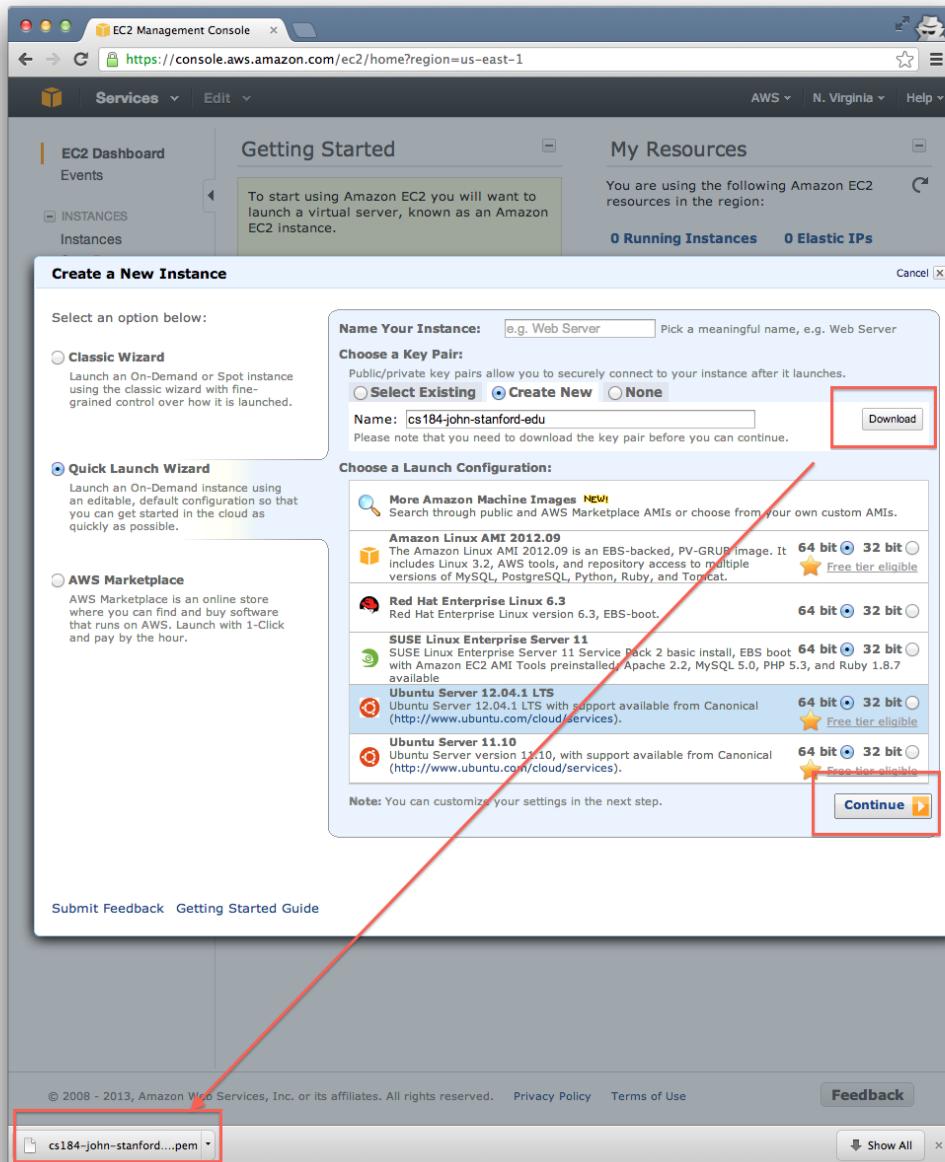


Figure 55: Click Download and notice the .pem file which you just created. Then click Continue.

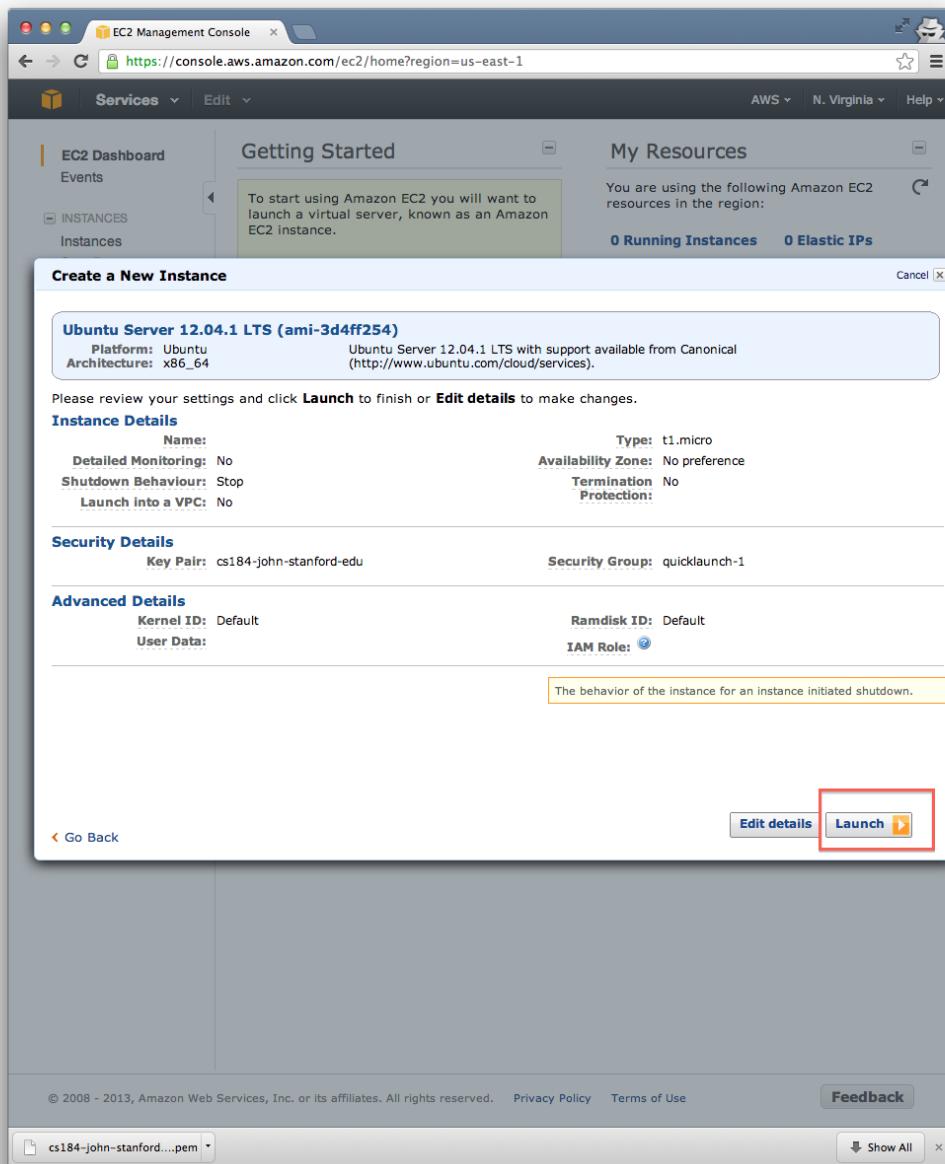


Figure 56: Launch the new instance.

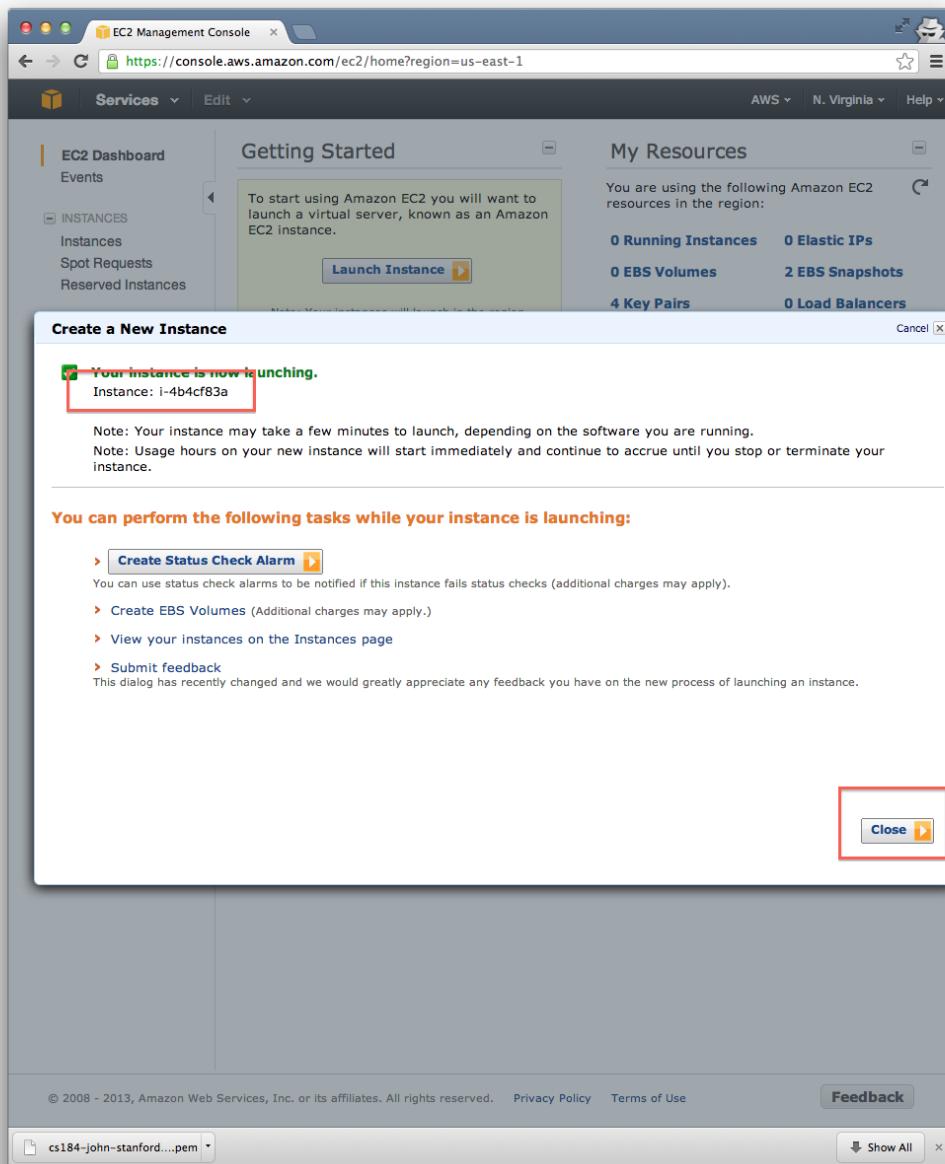


Figure 57: Your instance (in this case *i-4b4cf83a*) is now launching. Now go to the [Instances Dashboard](#).

When done initializing, you can connect to the EC2 instance.

Name	Instance	AMI ID	Root Device	Type	Status
empty	i-7dcfd602	ami-3d4ff254	ebs	t1.micro	terminated
empty	i-4b4cf83a	ami-3d4ff254	ebs	t1.micro	running initializing...

Figure 58: At the [Instances Dashboard](#), your instance (shown is `i-4b4cf83a` again) should be initializing. (Ignore the older terminated instance in this screenshot).

You should now have a local `.pem` file downloaded and the instance (in the above example `i-4b4cf83a`) should be in the process of initializing. You can monitor the status at this [dashboard link](#) (Figure 59). Now you want to connect to the instance, which in this case has hostname `ec2-50-19-140-229.compute-1.amazonaws.com` (yours will be different). For didactic purposes let's take a look at Amazon's instructions; we are going to modify these instructions slightly.

You can drag this pane upwards

Name	Instance	AMI ID	Root Device	Type	State	Status Checks
<input checked="" type="checkbox"/> empty	i-4b4cf83a	ami-3d4ff254	ebs	t1.micro	running	✓ 2/2 checks passed

EC2 Instance: i-4b4cf83a ●
ec2-50-19-140-229.compute-1.amazonaws.com

Description Status Checks Monitoring Tags

AMI: ubuntu/images/ebs/ubuntu-precise-12.04-amd64-server-20121001 (ami-3d4ff254)

Zone: us-east-1d

Type: t1.micro

Scheduled Events: No scheduled events

VPC ID: -

Source/Dest. Check: -

Placement Group: -

RAM Disk ID: -

Key Pair Name: cs184-john-stanford-edu

Monitoring: basic

Elastic IP: -

Alarm Status: none

Security Groups: quicklaunch-1, view rules

State: running

Owner: 896552222739

Subnet ID: -

Virtualization: paravirtual

Reservation: r-ca7b48b2

Platform: -

Kernel ID: aki-825ea7eb

AMI Launch Index: 0

Root Device: sda1

Figure 59: The *Instances Dashboard* will have a green check under “Status Checks” when the instance is live. Note that it is at this point that the hostname `ec2-50-19-140-229.compute-1.amazonaws.com` is assigned.

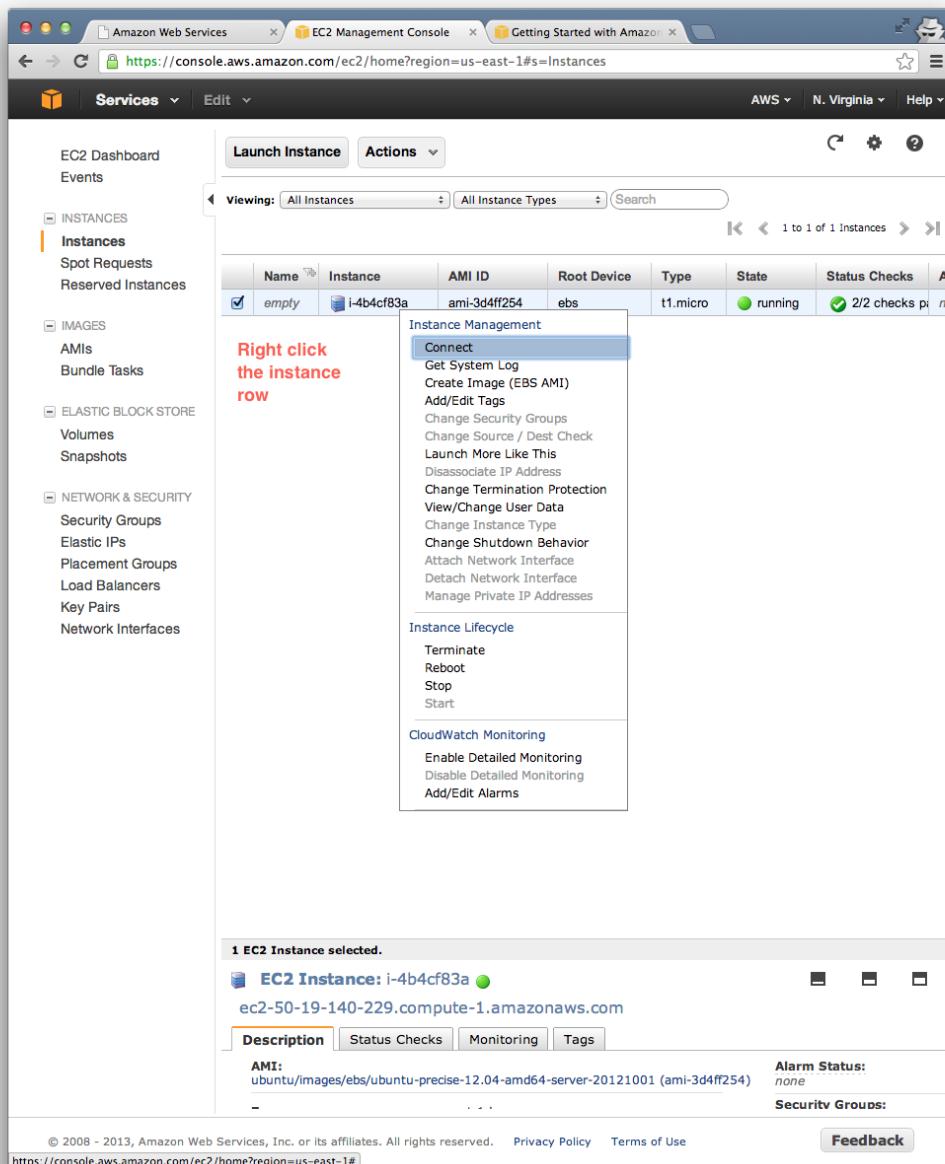


Figure 60: Right clicking on your instance and select ‘Connect’ from the dropdown.

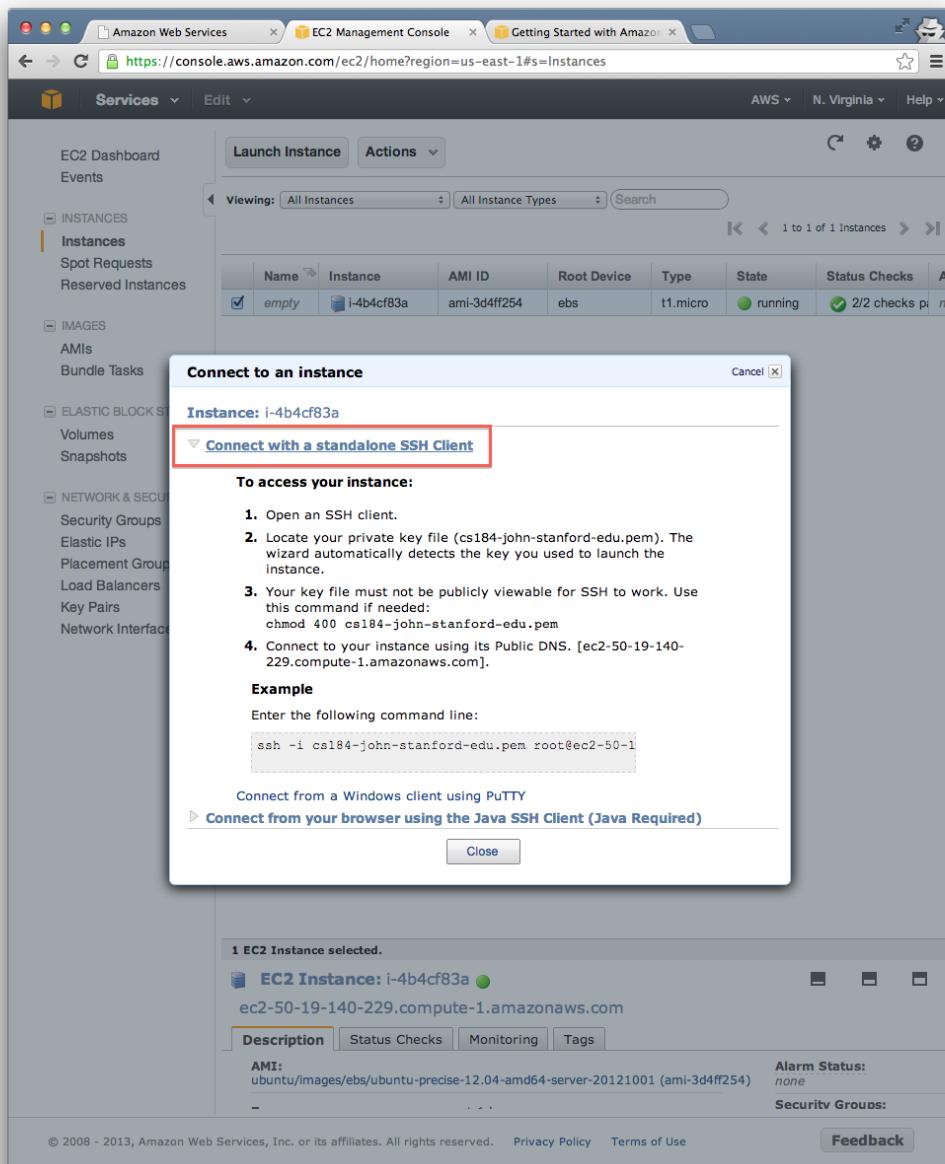


Figure 61: Click ‘Connect with a standalone SSH client’ and read the instructions. In the next section, we are going to have to use slightly different commands.

Mac: Connect to EC2 instance via Terminal.app

To connect to your EC2 instance via a Mac, type a modified version of the following commands into Terminal.app (see also Figure 62):

```
1 # Commands for Terminal.app on Mac OS X
2 $ cd ~/downloads
3 $ chmod 400 cs184-john-stanford-edu.pem
4 $ ssh -i cs184-john-stanford-edu.pem \
5     ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
```

Here, we use the standard convention that lines prefixed with # are comments and with \$ are meant to be typed in literally. When a backslash \ appears at the end of a line, that is a command which extends over multiple lines. Note also that you will need to change the .pem file (cs184-john-stanford-edu.pem) and hostname (ec2-50-19-140-229.compute-1.amazonaws.com) to match your own, and that we are logging in as `ubuntu` rather than `root`. Note in particular the use of the `chmod` command to change the permissions of your file. In the event you are using a multiuser system, like a shared computer in an academic environment, this would keep other users from logging in as you.

The screenshot shows a Terminal window on a Mac. The terminal session starts with:

```
[balajis@jiunit:~]$ cd downloads
[balajis@jiunit:~/downloads]$ chmod 400 cs184-john-stanford-edu.pem
[balajis@jiunit:~/downloads]$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
```

Following this, the terminal displays the Ubuntu welcome message and system information:

```
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

* Documentation: https://help.ubuntu.com/

System information as of Mon Jan  7 19:02:16 UTC 2013

System load:  0.0          Processes:      57
Usage of /:   11.5% of 7.87GB  Users logged in:  0
Memory usage: 30%          IP address for eth0: 10.211.171.76
Swap usage:   0%
```

It then provides links for system monitoring and package management:

```
Graph this data and manage this system at https://landscape.canonical.com/
65 packages can be updated.
31 updates are security updates.

Get cloud support with Ubuntu Advantage Cloud Guest
http://www.ubuntu.com/business/services/cloud
```

A red box highlights the command line from `chmod` to `ssh`, and a red arrow points from this box to a note on the right:

Note that we specify the username.

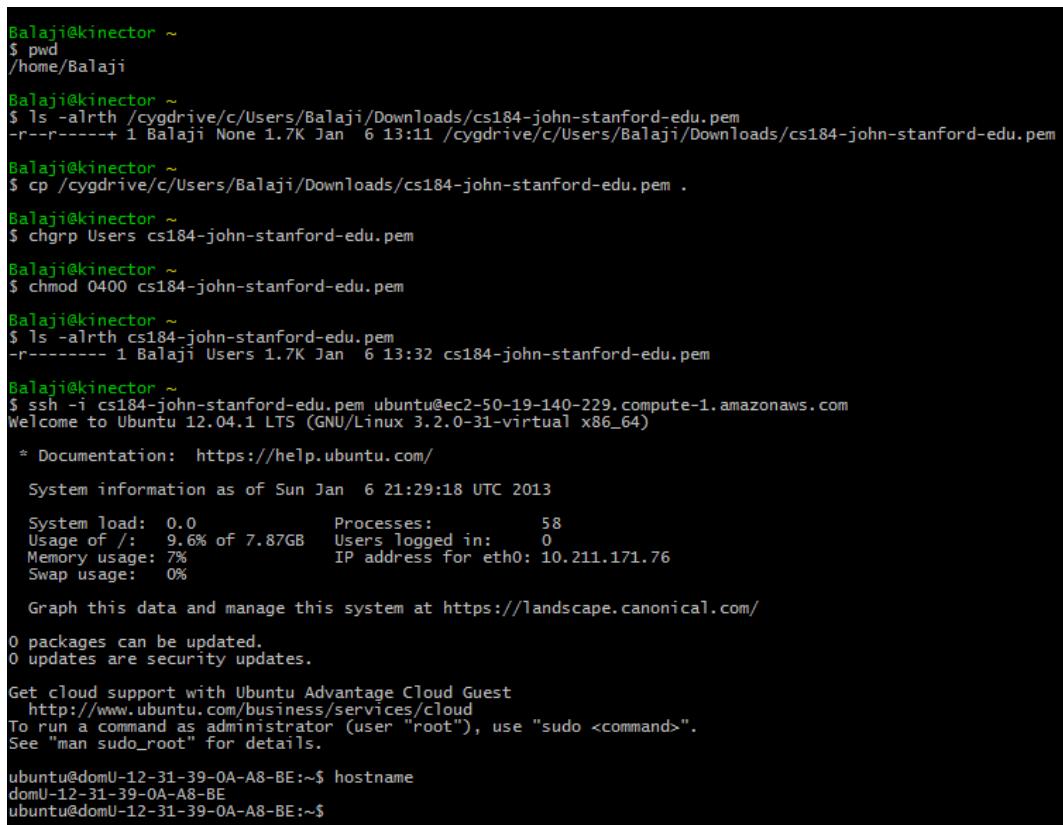
Figure 62: Connecting to your EC2 instance via Terminal.app on a Mac.

Windows: Connect to EC2 instance via Cygwin

To connect to EC2 via Cygwin on Windows, you need to handle two key idiosyncrasies. First, the Windows directory structure sits off the side from where Cygwin lives, so you need to address files in /cygdrive/c or /cygdrive/d to access files on the C:\ and D:\ drives, such as the .pem file you just downloaded. Second, Cygwin has a [permissions bug](#) which requires an extra command to fix. You will thus need to execute the following commands, replacing JohnSmith, cs184-john-stanford-edu.pem and ec2-50-19-140-229.compute-1.amazonaws.com with your own variables.

```
1 # Commands for Cygwin on Windows
2 $ cd ~
3 $ cp /cygdrive/c/Users/JohnSmith/Downloads/cs184-john-stanford-edu.pem .
4 $ chgrp Users cs184-john-stanford-edu
5 $ chmod 400 cs184-john-stanford-edu.pem
6 $ ssh -i cs184-john-stanford-edu.pem \
7     ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
```

Here is how all this looks when you launch Cygwin:



The screenshot shows a terminal window with a black background and white text. It displays a sequence of commands being entered and their execution results. The commands include navigating to the home directory, copying a .pem file from the Downloads folder to the current directory, changing the group ownership of the .pem file to 'Users', changing its mode to 400, and finally connecting via SSH to an EC2 instance using the copied pem file as the private key. The terminal also shows basic system information like CPU load, memory usage, and network status, followed by a welcome message from the Ubuntu 12.04.1 LTS system.

```
Balaji@kinector ~
$ pwd
/home/Balaji

Balaji@kinector ~
$ ls -alrth /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem
-r--r-----+ 1 Balaji None 1.7K Jan  6 13:11 /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem

Balaji@kinector ~
$ cp /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem .

Balaji@kinector ~
$ chgrp Users cs184-john-stanford-edu.pem

Balaji@kinector ~
$ chmod 0400 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ls -alrth cs184-john-stanford-edu.pem
-r----- 1 Balaji Users 1.7K Jan  6 13:32 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

 * Documentation:  https://help.ubuntu.com/
 * System information as of Sun Jan  6 21:29:18 UTC 2013
   System load:  0.0          Processes:      58
   Usage of /:  9.6% of 7.87GB  Users logged in:    0
   Memory usage: 7%           IP address for eth0: 10.211.171.76
   Swap usage:  0%
   Graph this data and manage this system at https://landscape.canonical.com/
 0 packages can be updated.
 0 updates are security updates.

Get cloud support with Ubuntu Advantage Cloud Guest
  http://www.ubuntu.com/business/services/cloud
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@domU-12-31-39-0A-A8-BE:~$ hostname
domU-12-31-39-0A-A8-BE
ubuntu@domU-12-31-39-0A-A8-BE:~$
```

Figure 63: Connecting to your EC2 instance via Cygwin for Windows.

```
Balaji@kinector ~
$ pwd
/home/Balaji

Balaji@kinector ~
$ ls -alrth /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem
-r--r-----+ 1 Balaji None 1.7K Jan  6 13:11 /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem

Balaji@kinector ~
$ cp /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem .

Balaji@kinector ~
$ chgrp Users cs184-john-stanford-edu.pem

Balaji@kinector ~
$ chmod 0400 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ls -alrth cs184-john-stanford-edu.pem
-r-----+ 1 Balaji Users 1.7K Jan  6 13:32 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

 * Documentation:  https://help.ubuntu.com/

 System information as of Sun Jan  6 21:29:18 UTC 2013

System load:  0.0      Processes:          58
Usage of /:   9.6% of 7.87GB  Users logged in:    0
Memory usage: 7%
Swap usage:   0%
IP address for eth0: 10.211.171.76

Graph this data and manage this system at https://landscape.canonical.com/

0 packages can be updated.
0 updates are security updates.

Get cloud support with Ubuntu Advantage Cloud Guest
http://www.ubuntu.com/business/services/cloud
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@domU-12-31-39-0A-A8-BE:~$ hostname
domU-12-31-39-0A-A8-BE
ubuntu@domU-12-31-39-0A-A8-BE:~$
```

Figure 64: The `chgrp` command changed the group membership of the `cs184-john-stanford-edu.pem` file from `None` to `Users`, as shown. (More on [Unix permissions](#).)

```

Balaji@kinector ~
$ pwd
/home/Balaji

Balaji@kinector ~
$ ls -alrth /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem
-r--r-----+ 1 Balaji None 1.7K Jan  6 13:11 /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem

Balaji@kinector ~
$ cp /cygdrive/c/Users/Balaji/Downloads/cs184-john-stanford-edu.pem .

Balaji@kinector ~
$ chgrp Users cs184-john-stanford-edu.pem

Balaji@kinector ~
$ chmod 0400 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ls -alrth cs184-john-stanford-edu.pem
-r----- 1 Balaji Users 1.7K Jan  6 13:32 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

 * Documentation: https://help.ubuntu.com/
 
 System information as of Sun Jan  6 21:29:18 UTC 2013

 System load:  0.0      Processes:           58
 Usage of `/': 9.6% of 7.87GB   Users logged in:     0
 Memory usage: 7%           IP address for eth0: 10.211.171.76
 Swap usage:   0%
 
 Graph this data and manage this system at https://landscape.canonical.com/
 
 0 packages can be updated.
 0 updates are security updates.

Get cloud support with Ubuntu Advantage Cloud Guest
 http://www.ubuntu.com/business/services/cloud
 To run a command as administrator (user "root"), use "sudo <command>".
 See "man sudo_root" for details.

ubuntu@domU-12-31-39-0A-A8-BE:~$ hostname
domU-12-31-39-0A-A8-BE
ubuntu@domU-12-31-39-0A-A8-BE:~$
```

Figure 65: The `chmod` command changed the read/write permissions of the `cs184-john-stanford-edu.pem` file such that only you can read it. (More on [Unix permissions](#).)

```

Balaji@kinector ~
$ chgrp None cs184-john-stanford-edu.pem

Balaji@kinector ~
$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
@@@@@@@@@@@ WARNING: UNPROTECTED PRIVATE KEY FILE! @@@@
Permissions 0440 for 'cs184-john-stanford-edu.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
bad permissions: ignore key: cs184-john-stanford-edu.pem
Permission denied (publickey).

Balaji@kinector ~
$ ls -alrth cs184-john-stanford-edu.pem
-r--r----- 1 Balaji None 1.7K Jan  6 13:32 cs184-john-stanford-edu.pem

Balaji@kinector ~
$ |
```

Figure 66: Here's what happens if your permissions are wrong: a public key error caused by incorrect permissions on the `.pem` file. Make sure you executed `chmod` and `chgrp`.

Unix permissions are a bit of a [deep topic](#), so you can just think of the above as an incantation for now that makes the files private to you. Note that if your permissions are incorrect (i.e. if you have not executed the `chmod 0400` and `chgrp Users` commands) you will get an error about public keys as shown above.

Security Groups

If you still have problems with the above commands, you might need to change your security groups to permit traffic on the SSH port.

The screenshot shows the AWS EC2 Management Console interface. On the left, there's a sidebar with links for EC2 Dashboard, Events, Instances (which is selected), Spot Requests, Reserved Instances, Images, AMIs, Bundle Tasks, Elastic Block Store, Volumes, Snapshots, Network & Security (Security Groups is selected), Elastic IPs, Placement Groups, Load Balancers, Key Pairs, and Network Interfaces. The main pane shows a table of instances with one row selected: 'empty' (Instance ID: i-4b4cf83a, AMI ID: ami-3d4ff254, Root Device: ebs, Type: t1.micro, State: running, Status Checks: initializing..., Alarm Status: none). Below the table, a message says '1 EC2 Instance selected.' followed by 'EC2 Instance: i-4b4cf83a'. To the right of this, a callout box contains the text 'Security Groups determine what can connect to your running instance.' A red arrow points from the 'Instances' link in the sidebar to the 'EC2 Instance selected' section. Another red arrow points to the 'Security Groups' field in the instance details, which is highlighted with a red box and contains the text 'quicklaunch-1. view rules'.

Figure 67: Select your instance and look at the pane on the lower right to see which security group you're running.

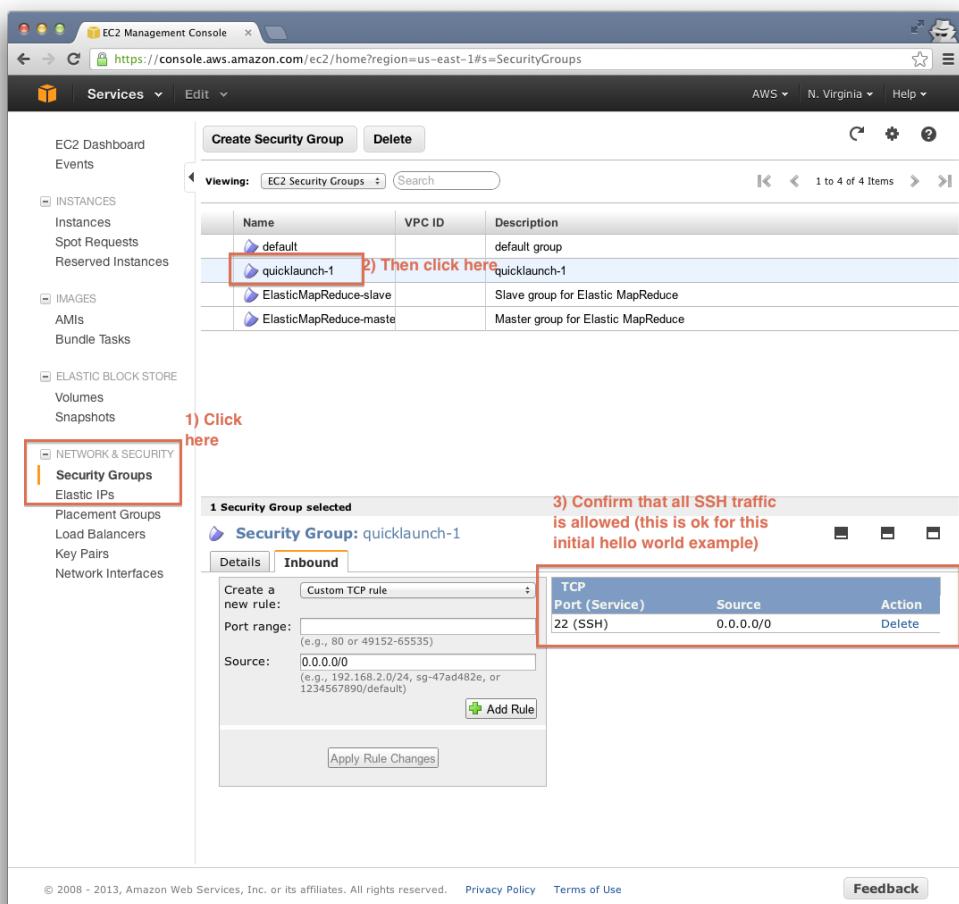


Figure 68: This is what things should look like if you click *Security Groups* (left hand side), your own security group at the top (which may or may not be titled *quicklaunch-1*), and then look in the bottom right corner.

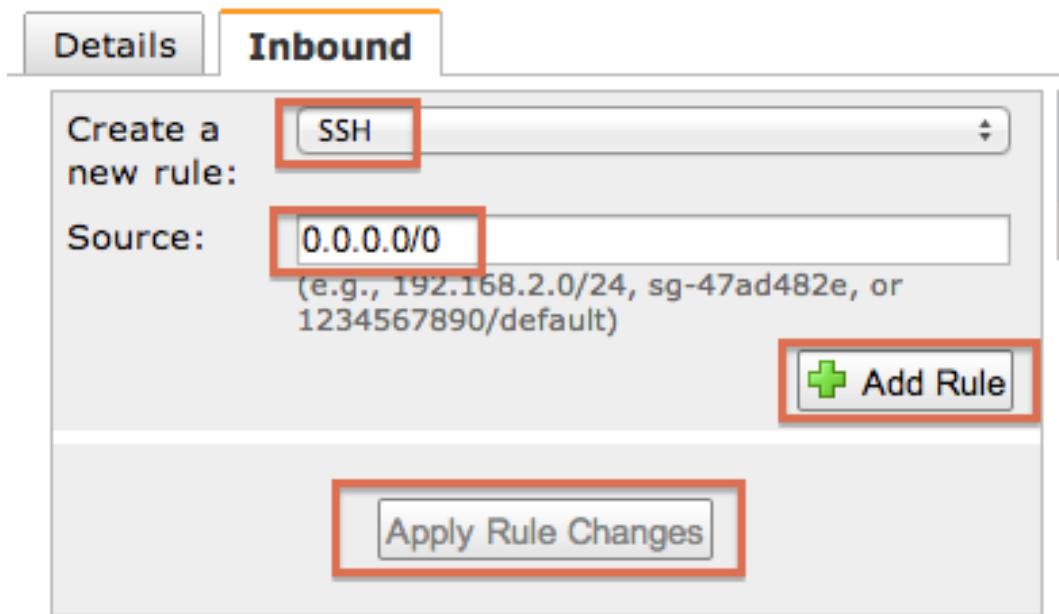


Figure 69: If necessary, use the form to open up SSH on port 22 with allowed IPs 0.0.0.0/0 (i.e. any IP), just as shown. You wouldn't do this in production, but it's fine for now.

Standard Operating System: Ubuntu 12.04.2 LTS on a t1.micro AWS instance

From this point forward, for the duration of the class unless otherwise specified, we assume that all commands are executed on your remote AWS cloud instance, specifically on Ubuntu 12.04.2 LTS on a t1.micro AWS instance (Figure 70). The idea here is to even out the dissimilarities between different computer setups and get an even (and reproducible) playing field for all students.

AWS Ubuntu 12.04.2 LTS Our common platform

No matter whether you have a Mac, Windows, or Linux machine, all further instructions will be given relative to the Amazon AMI running Ubuntu. Now everyone in the class has a common platform.

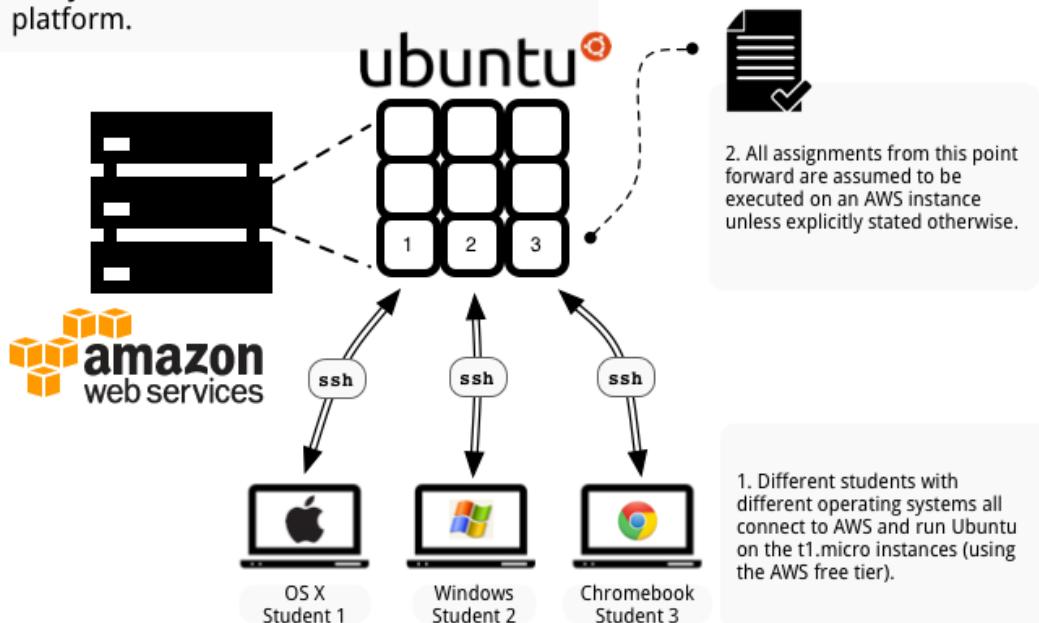


Figure 70: Note that AWS smooths out OS and configuration inhomogeneity. All assignments and commands from this point forward will be assumed to be run on a EC2 t1.micro instance running Ubuntu 12.04.2 LTS unless otherwise specified.

Going forward, note that conventionally the @ symbol is used to indicate where a command is being executed, to distinguish between remote vs. local machines. If you ever get confused, you can issue the command `hostname` to learn which machine you're on, as shown:

OK. You know that cloud thing everyone is talking about? You just did it. You rented a computer in the cloud and executed a few commands!

```
[balajis@jiunit:~/downloads]$hostname
jiunit
[balajis@jiunit:~/downloads]$echo "this command is executed on my local machine"
this command is executed on my local machine
[balajis@jiunit:~/downloads]$ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-54-234-41-176.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-40-virtual x86_64)

 * Documentation: https://help.ubuntu.com/

System information as of Wed Jun 19 13:20:30 UTC 2013

System load: 0.03      Processes:          61
Usage of /: 11.0% of 7.87GB  Users logged in:   1
Memory usage: 8%           IP address for eth0: 10.112.37.132
Swap usage:  0%
Graph this data and manage this system at https://landscape.canonical.com/

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

Use Juju to deploy your cloud instances and workloads:
https://juju.ubuntu.com/#cloud-precise

0 packages can be updated.
0 updates are security updates.

Last login: Wed Jun 19 13:20:05 2013 from 108-218-105-2.lightspeed.sntcca.sbcglobal.net
To run a command as administrator (user "root"), use "sudo <commands>".
See "man sudo_root" for details.

ubuntu@ip-10-112-37-132:~$ hostname
ip-10-112-37-132
ubuntu@ip-10-112-37-132:~$ echo "this command is executed on an AWS virtual machine"
this command is executed on an AWS virtual machine
ubuntu@ip-10-112-37-132:~$ exit
logout
Connection to ec2-54-234-41-176.compute-1.amazonaws.com closed.

[balajis@jiunit:~/downloads]$hostname
jiunit
[balajis@jiunit:~/downloads]$echo "i logged out. this command is executed on my local machine again."
i logged out. this command is executed on my local machine again.
[balajis@jiunit:~/downloads]$
```

Figure 71: Red are commands executed locally by your laptop. Blue are commands which are executed remotely by the AWS instance.

Deploy your code

Test out your Heroku account

We're now going to execute a series of commands that will deploy your first node.js app to heroku.

- First, you will SSH into your EC2 instance.
- Next, you'll install `git` and the `heroku` toolbelt.
- Next, login to heroku at the command line and set up SSH keys.
- Then, pull down a sample app, configure it as a heroku app, and push it live.
- Finally, view the app's URL in your browser.

That was in English. Here are the command lines to accomplish this:

```
1 # Execute these commands on your EC2 instance.
2 # Note that -q0- is not -q0-. 0 is the English letter, 0 is the number zero.
3
4 # 1) Install heroku and git
5 $ sudo apt-get install -y git-core
6 $ wget -q0- https://toolbelt.heroku.com/install-ubuntu.sh | sh
7 $ which git
8 $ which heroku
9 # 2) Login and set up your SSH keys
10 $ heroku login
11 $ ssh-keygen -t rsa
12 $ heroku keys:add
13 # 3) Clone a sample repo and push to heroku
14 $ git clone https://github.com/heroku/node-js-sample.git
15 $ cd node-js-sample
16 $ heroku create
17 $ git push heroku master
```

Now let's go through the screenshots of how it looks when executing those commands. Again, begin by connecting to your EC2 instance.

The screenshot shows a terminal window on an Ubuntu 12.04.1 LTS system. The user has SSHed into their EC2 instance using a private key. The terminal output includes system information, package updates, and the execution of an apt-get command to install the 'git-core' package. A red box highlights the command 'sudo apt-get install git-core'. Another red box highlights the question '[Y/n]? Y' at the end of the installation process.

```

[balajis@junit:~]$ls
[balajis@junit:~/downloads]$ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-29.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

 * Documentation:  https://help.ubuntu.com/
 
 System information as of Mon Jan  7 00:24:25 UTC 2013

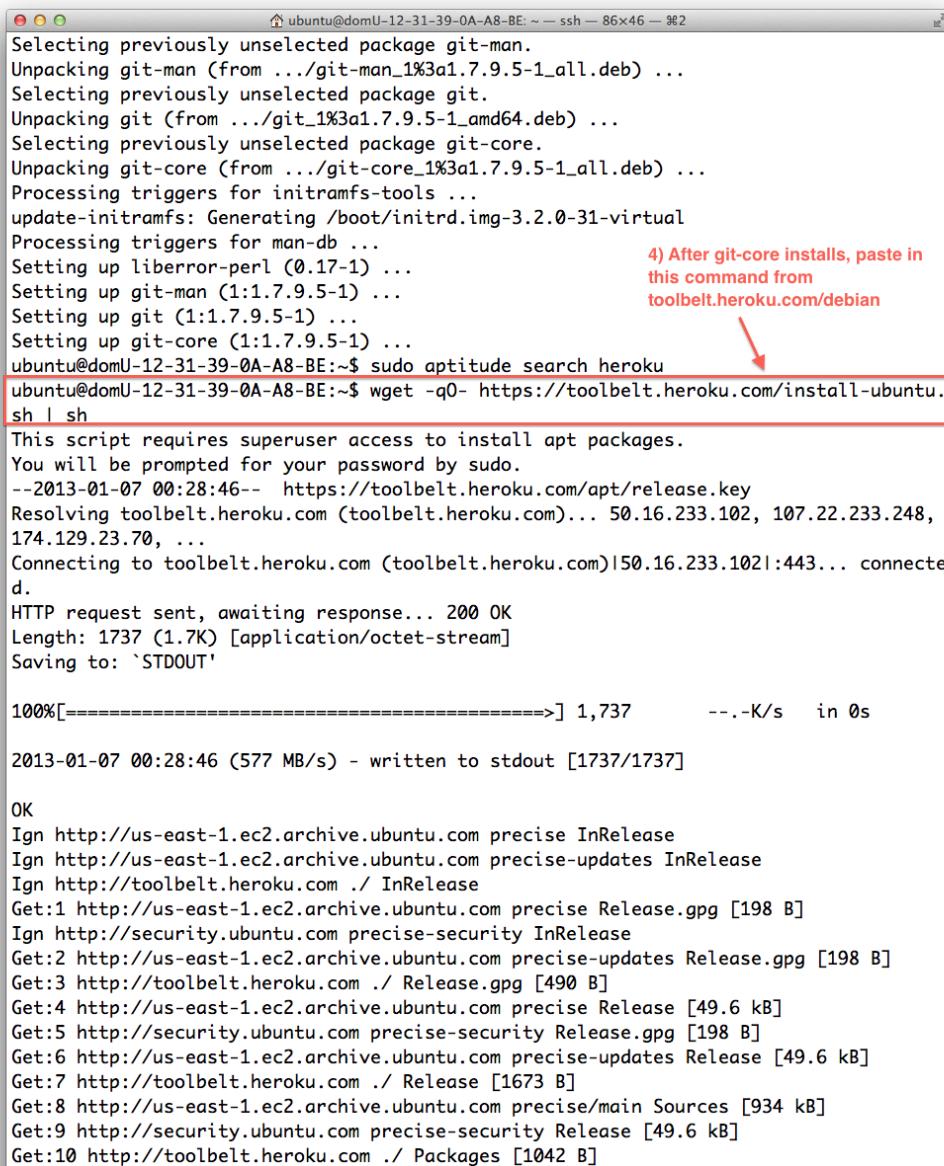
 System load:  0.0          Processes:      58
 Usage of /:   9.6% of 7.87GB  Users logged in:    0
 Memory usage: 7%           IP address for eth0: 10.211.171.76
 Swap usage:   0%
 
 Graph this data and manage this system at https://landscape.canonical.com/
 
 0 packages can be updated.
 0 updates are security updates.

 Get cloud support with Ubuntu Advantage Cloud Guest
 http://www.ubuntu.com/business/services/cloud
 To run a command as administrator (user "root"), use "sudo <command>".
 See "man sudo_root" for details.

ubuntu@domU-12-31-39-0A-A8-BE:~$ which git
ubuntu@domU-12-31-39-0A-A8-BE:~$ sudo aptitude search git-core
p  git-core                               - fast, scalable, distributed revision control
v  git-core:i386                            -
ubuntu@domU-12-31-39-0A-A8-BE:~$ sudo apt-get install git-core
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  git git-man liberror-perl
Suggested packages:
  git-daemon-run git-daemon-sysvinit git-doc git-el git-arch git-cvs git-svn
  git-email git-gui gitk gitweb
The following NEW packages will be installed:
  git git-core git-man liberror-perl
0 upgraded, 4 newly installed, 0 to remove and 0 not upgraded.
Need to get 6742 kB of archives.
After this operation, 15.3 MB of additional disk space will be used.
Do you want to continue [Y/n]? Y  3) Say yes

```

Figure 72: SSH into your EC2 instance and install `git-core` as shown. If you do `sudo apt-get install -y git-core` you won't have to type the `Y`.



The screenshot shows a terminal window on an Ubuntu system. The terminal output is as follows:

```

ubuntu@domU-12-31-39-0A-A8-BE: ~ ssh 86x46 9%2
Selecting previously unselected package git-man.
Unpacking git-man (from .../git-man_1%3a1.7.9.5-1_all.deb) ...
Selecting previously unselected package git.
Unpacking git (from .../git_1%3a1.7.9.5-1_amd64.deb) ...
Selecting previously unselected package git-core.
Unpacking git-core (from .../git-core_1%3a1.7.9.5-1_all.deb) ...
Processing triggers for initramfs-tools ...
update-initramfs: Generating /boot/initrd.img-3.2.0-31-virtual
Processing triggers for man-db ...
Setting up liberror-perl (0.17-1) ...
Setting up git-man (1:1.7.9.5-1) ...
Setting up git (1:1.7.9.5-1) ...
Setting up git-core (1:1.7.9.5-1) ...
ubuntu@domU-12-31-39-0A-A8-BE:~$ sudo aptitude search heroku
ubuntu@domU-12-31-39-0A-A8-BE:~$ wget -qO- https://toolbelt.heroku.com/install-ubuntu.sh | sh
This script requires superuser access to install apt packages.
You will be prompted for your password by sudo.
--2013-01-07 00:28:46-- https://toolbelt.heroku.com/apt/release.key
Resolving toolbelt.heroku.com (toolbelt.heroku.com)... 50.16.233.102, 107.22.233.248,
174.129.23.70, ...
Connecting to toolbelt.heroku.com (toolbelt.heroku.com)|50.16.233.102|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1737 (1.7K) [application/octet-stream]
Saving to: `STDOUT'

100%[=====] 1,737      --.-K/s   in 0s

2013-01-07 00:28:46 (577 MB/s) - written to stdout [1737/1737]

OK
Ign http://us-east-1.ec2.archive.ubuntu.com precise InRelease
Ign http://us-east-1.ec2.archive.ubuntu.com precise-updates InRelease
Ign http://toolbelt.heroku.com ./ InRelease
Get:1 http://us-east-1.ec2.archive.ubuntu.com precise Release.gpg [198 B]
Ign http://security.ubuntu.com precise-security InRelease
Get:2 http://us-east-1.ec2.archive.ubuntu.com precise-updates Release.gpg [198 B]
Get:3 http://toolbelt.heroku.com ./ Release.gpg [490 B]
Get:4 http://us-east-1.ec2.archive.ubuntu.com precise Release [49.6 kB]
Get:5 http://security.ubuntu.com precise-security Release.gpg [198 B]
Get:6 http://us-east-1.ec2.archive.ubuntu.com precise-updates Release [49.6 kB]
Get:7 http://toolbelt.heroku.com ./ Release [1673 B]
Get:8 http://us-east-1.ec2.archive.ubuntu.com precise/main Sources [934 kB]
Get:9 http://security.ubuntu.com precise-security Release [49.6 kB]
Get:10 http://toolbelt.heroku.com ./ Packages [1042 B]

```

A red box highlights the command `sh | sh`. A red arrow points from the text "4) After git-core installs, paste in this command from toolbelt.heroku.com/debian" to the highlighted command.

Figure 73: Once the installation of `git-core` completes, paste in the command from toolbelt.heroku.com/debian to install the `heroku` command line tools.

```

ubuntu@domU-12-31-39-0A-A8-BE:~$ which git
/usr/bin/git
ubuntu@domU-12-31-39-0A-A8-BE:~$ which heroku
/usr/bin/heroku
ubuntu@domU-12-31-39-0A-A8-BE:~$ which foreman
/usr/bin/foreman
ubuntu@domU-12-31-39-0A-A8-BE:~$ git --version
git version 1.7.9.5

```

5) You can confirm that *git* and *heroku* were installed properly by running the "which" command. It should produce the paths as shown.

Figure 74: Now confirm that *git* and *heroku* are installed and on your path. If they were not, then *which* would not print anything. If they are both installed, *which* will print the location of the programs, as shown they are both in */usr/bin*.

```

ubuntu@domU-12-31-39-0A-A8-BE:~/node-js-sample$ heroku login
Enter your Heroku credentials.
Email: balajis@stanford.edu
Password (typing will be hidden):
Authentication successful.

```

6) Log in at the command line with the same user/pass as heroku.com

Figure 75: Log in to heroku from the command line, using the same user/pass you set up at heroku.com.

```

ubuntu@domU-12-31-39-0A-A8-BE:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
5f:7b:87:19:0c:c8:23:aa:06:97:77:12:52:27:81:12 ubuntu@domU-12-31-39-0A-A8-BE
The key's randomart image is:
+--[ RSA 2048]--+
| E. .+.. |
| . . . o . . |
| . . . + . . |
| o o . . o |
| . o + S . o |
| o o o . . . + |
| o . . . + . |
| . . . . . |
+-----+
ubuntu@domU-12-31-39-0A-A8-BE:~$

```

7) Next we set up your SSH keys for talking to Heroku. Just type in this command and hit enter twice ("empty for no passphrase")

Figure 76: Here we are setting up your SSH keys for connecting to Heroku. Just hit enter twice when prompted for the passphrase.

```

ubuntu@domU-12-31-39-0A-A8-BE:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
5f:7b:87:19:0c:c8:23:aa:06:97:77:12:52:27:81:12 ubuntu@domU-12-31-39-0A-A8-BE
The key's randomart image is:
+--[ RSA 2048]----+
| E. .+.. |
| . . o . . |
| . . . + . |
| o o . . o |
| . o + S . o |
| o o o . . + |
| o . . . + . |
| . . . . . |
+-----+
ubuntu@domU-12-31-39-0A-A8-BE:~$ heroku keys:add
Found existing public key: /home/ubuntu/.ssh/id_rsa.pub
Uploading SSH public key /home/ubuntu/.ssh/id_rsa.pub... done

```

8) After you've done the ssh-keygen step, now tell heroku about the new keys by typing in this command.



Figure 77: Now add the SSH keys you just generated to heroku (specifically you are sending the public key to heroku).

```

ubuntu@domU-12-31-39-0A-A8-BE:~$ git clone https://github.com/heroku/node-js-sample.git
Cloning into 'node-js-sample'...
remote: Counting objects: 277, done.
remote: Compressing objects: 100% (240/240), done.
remote: Total 277 (delta 9), reused 276 (delta 9)
Receiving objects: 100% (277/277), 191.16 KiB, done.
Resolving deltas: 100% (9/9), done.
ubuntu@domU-12-31-39-0A-A8-BE:~$ cd node-js-sample/
ubuntu@domU-12-31-39-0A-A8-BE:~/node-js-sample$ heroku create
Enter your Heroku credentials.
Email: balajis@stanford.edu
Password (typing will be hidden):
Creating guarded-lake-3837... done, stack is cedar
http://guarded-lake-3837.herokuapp.com/ | git@heroku.com:guarded-lake-3837.git
Git remote heroku added
ubuntu@domU-12-31-39-0A-A8-BE:~/node-js-sample$

```

8) Execute this command to pull down the sample nodejs app. Note the trailing t on the next line.

9) After the cloning completes, cd into the node-js-sample directory.

10) Enter this command to set up a heroku app with a random name (e.g. guarded-lake-3837 in this example) using the code in this git repository.

Figure 78: Type in the commands shown to pull down the sample nodejs codebase and create a corresponding heroku app for that codebase.

```

ubuntu@domU-12-31-39-0A-A8-BE:~$ cd node-js-sample/
ubuntu@domU-12-31-39-0A-A8-BE:~/node-js-sample$ git push heroku master
Counting objects: 277, done.
Compressing objects: 100% (240/240), done.
Writing objects: 100% (277/277), 191.16 KiB, done.
Total 277 (delta 9), reused 277 (delta 9)
----> Node.js app detected
----> Resolving engine versions
      Using Node.js version: 0.8.14
      Using npm version: 1.1.65
----> Fetching Node.js binaries
----> Vendorizing node into slug
----> Installing dependencies with npm
      npm WARN package.json node-example@0.0.1 No README.md file found!
      npm WARN package.json node-example@0.0.1 No README.md file found!
      npm WARN package.json connect@1.9.2 No README.md file found!
      express@2.5.11 /tmp/build_2znwj0puqx52w/node_modules/express
      connect@1.9.2 /tmp/build_2znwj0puqx52w/node_modules/express/node_modules/connect
t
      qs@0.4.2 /tmp/build_2znwj0puqx52w/node_modules/express/node_modules/qs
      mime@1.2.4 /tmp/build_2znwj0puqx52w/node_modules/express/node_modules/mime
      formidable@1.0.11 /tmp/build_2znwj0puqx52w/node_modules/express/node_modules/connect/node_modules/formidable
      mkdirp@0.3.0 /tmp/build_2znwj0puqx52w/node_modules/express/node_modules/mkdirp
      Dependencies installed
----> Building runtime environment
----> Discovering process types
      Procfile declares types -> web
----> Compiled slug size: 3.8MB
----> Launching... done, v4
      http://guarded-lake-3837.herokuapp.com deployed to Heroku

```

11) Finally, enter this command, also within the node-js-sample directory, to push the code to the app you just created.

12) Awesome. We now have a URL up and live. You will see a different app name - go and enter this into your browser now.

Figure 79: Last step: type in `git push heroku master` and then confirm that a URL is printed at the end (something like `http://guarded-lake-3837.herokuapp.com`, but yours will be different.)

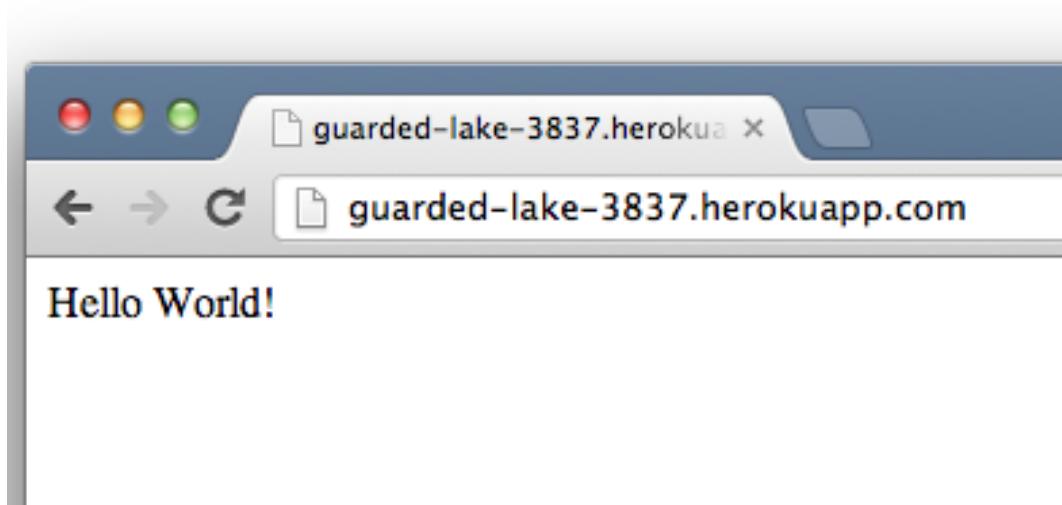


Figure 80: Your first heroku nodejs app is now online! Your URL will be similar to `http://guarded-lake-3837.herokuapp.com`, but you will have your own custom version.

Congratulations! You just launched a cloud computer via AWS' EC2, cloned code from Github, and deployed a very simple website on Heroku. The last thing you want to do is terminate your EC2 instance so that it doesn't keep running.

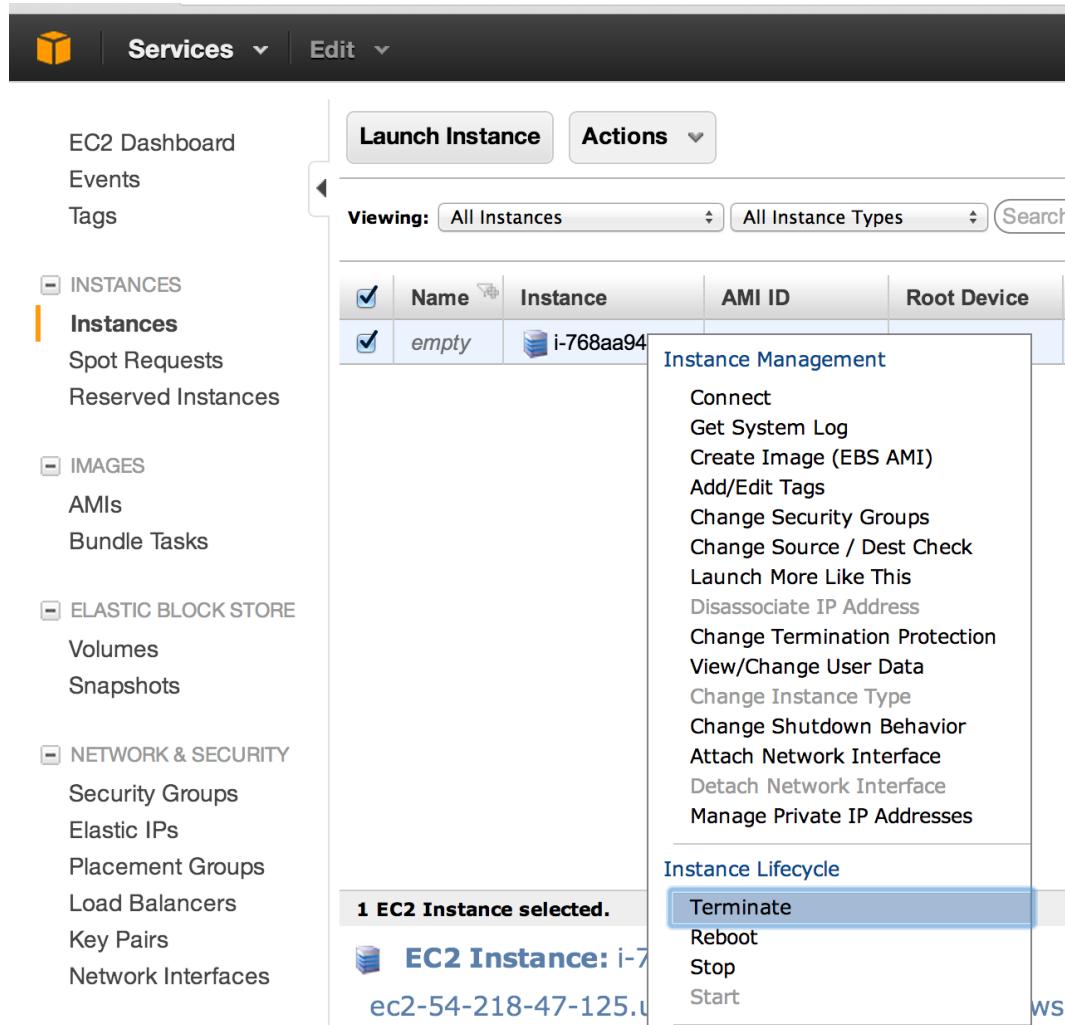


Figure 81: After you've gotten your Heroku app live, go to the *Instances Dashboard*, right-click your instance, and select *Terminate*.

Note: it is also possible to stop an instance. Stopping is different from terminating; while terminating completely kills and frees up all resources, stopping an EBS-backed instances pauses billing for the computing portion but still charges for the storage. We'll get into this later, but read [here](#) and [here](#) for more.

Post-mortem

Now that you have your site up, take a look again at Figure 82. We did most of the things in there, except we only pulled code from [github.com](#) and didn't push any code. It's very important to keep in mind which computer you are currently controlling and executing commands on, whether that be your own laptop, an AWS machine, or a remote machine in the Heroku or Github clouds. Over time this will become second nature, and we'll show how to configure

your command line environment to give some hints.

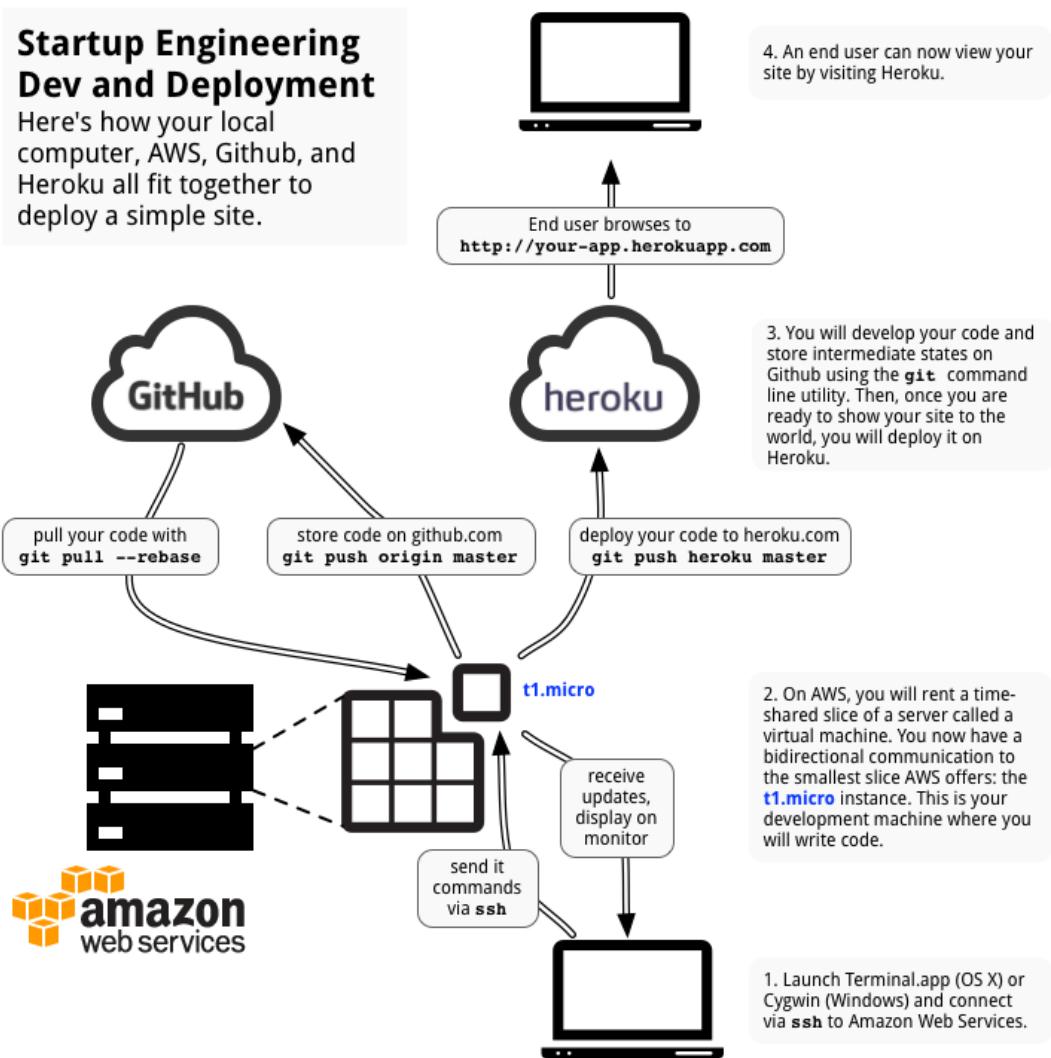


Figure 82: Recall this figure again. You were controlling the **t1.micro** instance as you typed in those commands to pull down code from github and push it to heroku.

In the interests of speed, we did a number of things in this interactive start that we would not normally do. For example, we worked with `.pem` files in the directory where they landed, rather than moving them into `~/.ssh`. And in practice you would want to put code through three stages: a dev, staging, and production branch before pushing to the web. You probably also want to read the Heroku documentation on [node.js](#) and [SSH keys](#) to understand some of the steps at the end. We'll cover things like this in more detail as the course progresses.

Terminology

Whew! We flew through quite a bit there. Here are a few terms which we introduced that you should become familiar with:

- **git**: A tool for storing and manipulating the history of a program edited by many different users. Also known as a distributed version control system (DVCS).
- *Github.com*: A social network for programmers built around **git**. There are other sites like this, such as BitBucket.com and Sourceforge.com
- **AWS**: Amazon Web Services, a set of cloud services for programmers.
- **Heroku**: A layer on top of AWS that greatly simplifies the process of developing and deploying a web application.
- **Gravatar**: A simple site which provides a globally recognizable image that you can use in various social contexts. For example, click [here](#) and look at the little images to the left of each **git** commit; those are being pulled from gravatar.com.
- **EC2**: Elastic Compute Cloud, a fleet of virtual machines that you can rent by the hour from Amazon.
- **Virtual machine**: In practice, you are usually not renting a single physical computer, but rather a virtualized piece of a multiprocessor computer. Breakthroughs in operating systems research have allowed us to take a single computer with (say) eight processors and make it seem like eight independent computers, capable of being completely wiped and rebooted without affecting the others.

A good analogy for virtualization is sort of like taking a house with eight rooms and converting it into eight independent apartments, each of which has its own key and climate control, and can be restored to default settings without affecting the others.

While these “virtual computers” have reasonable performance, they are less than that of the native hardware. In practice, virtualization is not magic; extremely high workloads on one of the other “tenants” in the VM system can affect your room, in the same way that a very loud next door neighbor can overwhelm the soundproofing. Still, most of the time you don’t need an eight room home or an eight processor computer; you need just one room for the night.

- **AMI**: Amazon Machine Image, an operating system + default software package which you can use to initialize any instance.
- **Availability Zone**: the physical location of the instance.
- **EC2 instance**: a particular instance, e.g. `i-2938408`.
- **IP address**: a dotted address representing a machine on the internet, e.g. `171.65.1.2`.
- **Hostname**: The name of a server, e.g. `ec2-1234.amazon.com`.
- **DNS**: The system which maps a hostname like `ec2-1234.amazon.com` to an IP address like `171.65.1.2`.

- *SSH*: Secure Shell, a means of connecting to a remote computer and execute commands.
- *Public Key*: When you hand out your public key, it is like handing out a set of special envelopes. Any who has your public key can put their message in one of these envelopes, seal it, and send it to you. But only you can open that message with your private key. Both of the “keys” in this case can be thought of as long hexadecimal numbers. Github asks for your public key to confirm your identity in future interactions; essentially for all future communications they send you a message that only you can open (with your private key). By decrypting this message and sending it back to them, they know that you must have the private key and so they authorize the operation.
- *RSA*: A kind of [encryption algorithm](#) that generates both public and private keys. Always run this locally on your machine; anyone who knows your private key can decrypt messages encrypted with the corresponding public key and thereby pose as you.

Linux and Server-Side Javascript (SSJS)

Overview

You are likely a user of the Windows or OS X operating systems. As you may know, the operating system (OS) interfaces with a computer's hardware (like the CPU, hard drive, and network card) and exposes it via API to running programs. The OS layer thus allows an engineer to write bytes to a hard drive or send packets over a network card without worrying about the underlying details.

Most end users interact with their operating system via a graphical user interface (GUI) to watch movies, surf the web, send email, and the like. However, when your goal is to *create* new programs rather than simply *run* them, you will want to use an OS which is primarily controlled by the command-line interface (CLI), usually a Unix-based OS. Roughly speaking, Windows is mostly used for its GUI features, Linux mostly for its command line features, and OS X (built on BSD) for both. The convention in Silicon Valley today is to use OS X machines for local development and daily use, with Linux machines for production deployments and Windows machines only for testing IE or interfacing with certain kinds of hardware (e.g. Kinects). By combining OS X and Linux in this fashion one can get the best of both worlds: a modern desktop interface and a command line locally via OSX, with a license-free and agile server OS via Linux.

Now, before we go on it's worth drawing a distinction between [Unix](#), [Linux](#), [BSD](#), and [OS X](#). Unix is the granddaddy of them all, going back to AT&T. BSD is a [lineal descendant](#) of the original Unix, while [Linux began](#) as an API-compatible clone that had no shared code with the original Unix. And Mac OS X is a proprietary OS built on top of BSD which adds many new features. Today there are an [incredible number](#) of Linux and BSD variants, and you can trace through the evolution of the operating systems [here](#). For the purposes of this class, we'll be using the Linux distribution called [Ubuntu](#) for development.

Key Features of Linux

In general, we're not going to get into low-level Linux details here, though you will [need to](#) if your app becomes wildly successful or uses significant hardware resources. The main things you need to know about Linux are:

1. *Command Line Interface*: Linux variants are built to be controlled from the command line. Every single thing that you need to accomplish can be done by typing into a command line interface (CLI) rather than clicking on buttons in a graphical user interface (GUI). In particular, it is possible to control a Linux box simply by typing in commands over a low-bandwidth text connection rather than launching a full-blown windowing system with image backgrounds and startup music.
2. *No Licensing Fees*: Linux does not require paying licensing fees to install. In theory, you can run a fleet of 1000 machines without paying any vendor fees, unlike Windows.

In practice you will need to pay for maintenance in some fashion, either via support agreements, in-house sysadmins, or by renting the computers from a cloud provider.

3. *Open Source*: Linux has a massive base of open-source software which you can download for free, mostly geared towards the needs of engineers. For example, you will be able to download free tools to edit code, convert images, and extract individual frames from movies. However, programs for doing things like *viewing* movies will not be as easy to use as the Microsoft or Apple equivalents.
4. *Server-side Loophole*: Linux is licensed under the [GPL](#), one of the major open source licenses. What this means is that if you create and distribute a modified version of Linux, you need to make the source code of your modifications available for free. The GPL has been called a viral license for this reason: if you include GPL software in your code, and then offer that software for download, then you must make the full source code available (or risk a lawsuit from the [EFF](#)). There are workarounds for this viral property, of which the most important is called the server-side or application-service provider (ASP) loophole: as long as your software is running behind a web server, like Google, the GPL terms do not consider you a *distributor* of the code. That is, Google is only offering you a search box to use; it's not actually providing the full source code for its search engine for download, and is thus not considered a distributor. A new license called the [AGPL](#) was written to close this loophole, and there are some major projects like [MongoDB](#) that use it.
5. *Linux and Startups*: The ASP loophole is what allows modern internet companies to make a profit while still using Linux and free software. That is, they can make extensive use of open source software on their servers, without being forced to provide that software to people using their application through the browser. In practice, many internet companies nevertheless give back to the community by [open sourcing](#) significant programs or large portions of their code base. However, it should be clear that the server-side exception is a structural disadvantage for traditional “shrink-wrapped” software companies like Microsoft, for whom the GPL was originally designed to target. By distributing software on a CD or DVD, you lose the ability to use open source code, unless you do something like making the distributed software [phone home](#) to access the open source portion of the code from one of your servers. It should also be noted that another way to make money with open source is the [professional open source](#) business model pioneered by RedHat; the problem with this kind of model is that a service company must recruit talented people to scale up (a stochastic and non-reproducible process, subject to diminishing returns), while a product company only requires more servers to scale up (which is much more deterministic). As such the profitability and attractiveness of a service company as a VC investment is inherently limited.
6. *OS X client, Linux server*: The most common way to use Linux in Silicon Valley today is in combination with OS X. Specifically, the standard setup is to equip engineers with Macbook Pro laptops for local development and use servers running Linux for production deployment. This is because Apple’s OS X runs BSD, a Unix variant which is similar enough to Linux to permit most code to run unchanged on both your local OS X laptop and the remote Linux server. An emerging variant is one in which [Chromebooks](#) are used as the local machines with the [Chrome SSH client](#) used to connect to a remote Linux development server. This is cheaper from a hardware perspective (Chromebooks

are only \$200, while even Macbook Airs are \$999) and much more convenient from a system administration perspective (Chromebooks are hard to screw up and easy to wipe), but has the disadvantage that Chromebooks are less useful offline. Still, this variant is worth considering for cash-strapped early stage startups.

7. *Ubuntu: desktop/server Linux distribution:* Perhaps the [most popular](#) and all-round polished desktop/server Linux distribution today is [Ubuntu](#), which has a very convenient package management system called [apt-get](#) and significant funding for polish from [Mark Shuttleworth](#). You can get quite far by learning Ubuntu and then assuming that the concepts map to other Linux distributions like [Fedora/RedHat/CentOS](#).
8. *Android: mobile Linux distribution:* One last point to keep in mind is that Google's Android OS is also [Linux-based](#) and undeniably has many more installed units worldwide than Ubuntu; however, these units are generally smartphones and tablets rather than desktops, and so one usually doesn't think of Android in the same breath as Ubuntu for an engineer's primary development OS (as opposed to a deployment OS used by consumers). That may change, however, as [Android on Raspberry Pi](#) continues to gain steam and Pi-sized units become more powerful.

Virtual Machines and the Cloud

The Concept of Virtualization

When you rent a computer from AWS or another infrastructure-as-a-service (IAAS) provider, you are usually not renting a single physical computer, but rather a virtualized piece of a multiprocessor computer. Breakthroughs in operating systems research (in large part from [VMWare](#)'s founders, both from Stanford) have allowed us to take a single computer with eight processors and make it seem like eight independent computers, each capable of being completely wiped and rebooted without affecting the others. A good analogy for virtualization is sort of like taking a house with eight rooms and converting it into eight independent hotel rooms, each of which has its own 24-hour access and climate control, and can be restored to default settings by a cleaning crew without affecting the other rooms.

Virtualization is extremely popular among corporate IT groups as it allows maximum physical utilization of expensive multiprocessor hardware (which has become the standard due to the [flattening of Moore's law on single CPUs](#)). Just like a single person probably won't make full use of an eight-room house, a single 8-CPU server is usually not worked to the limit 24/7. However, an 8-CPU computer split into 8 different virtual servers with different workloads often has much better utilization.

While these "virtual computers" have reasonable performance, they are less than that of the native hardware. And in practice, virtualization is not magic; extremely high workloads on one of the other "tenants" in the VM system can affect your room, in the same way that a very loud next-door neighbor can overwhelm the soundproofing (see the remarks on "Multi-Tenancy" in this [blog post](#)). Still, for the many times you don't need an eight-room home or an eight-processor computer, and just need one room/computer for the night, virtualization is very advantageous.

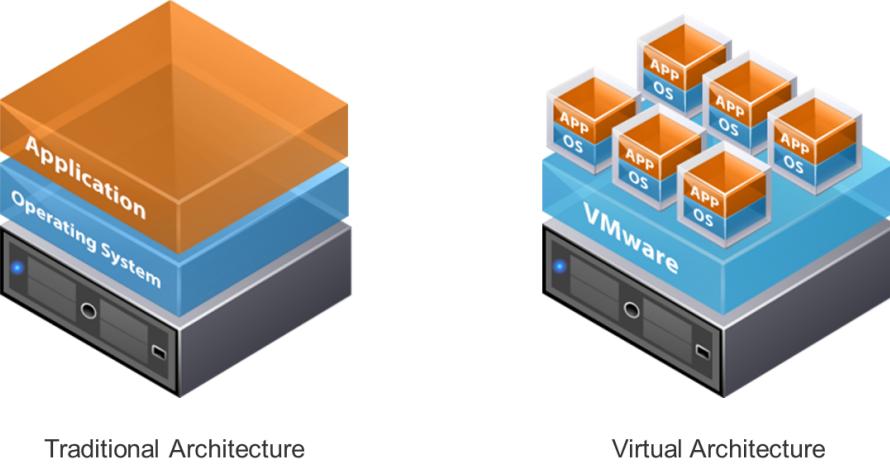


Figure 1: The concept of virtualization, as illustrated by a figure from [VMWare](#). A single piece of physical hardware can run several different virtual machines, each of which appears to the end user like a full-fledged piece of physical hardware. Individual virtual machines can be shut down or have their (virtual) hard drives reformatted without affecting the uptime of the host OS.

The Cloud and IAAS/PAAS/SAAS

One definition of a *cloud computer* is a computer whose precise physical location is immaterial to the application. That is, you don't need to have it be present physically to make something happen; it just needs to be accessible over a (possibly wireless) network connection. In practice, you can often determine where a cloud computer is via commands like [dig](#) and [ping](#) to estimate latency, and there are applications (particularly in real-time communication, like instant messaging, gaming, telephony, or telepresence) where the speed of light and latency correction is a nontrivial consideration (see this [famous post](#)).

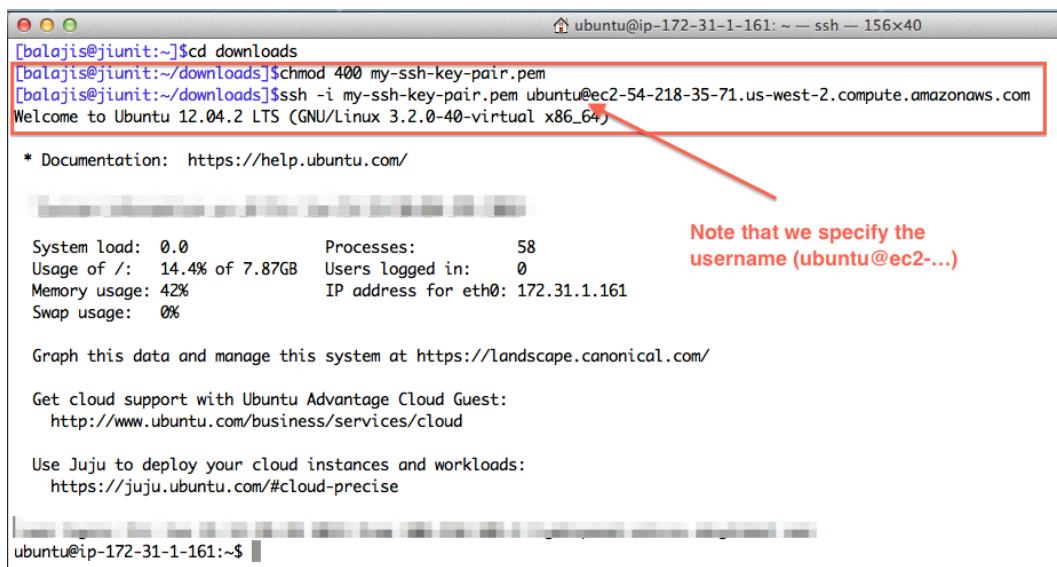
However, for the large number of applications that can tolerate some (small) degree of latency and/or physical separation from a computer, cloud computing is quite useful. There are [three classes](#) of cloud computing, broadly speaking:

1. [IAAS](#): AWS, Joyent, Rackspace Cloud
2. [PAAS](#): Heroku, DotCloud, Nodester, Google AppEngine
3. [SAAS](#): Salesforce, Google Apps, Mint.com

With IAAS you get direct command line access to the hardware, but have to take care of all the details of actually deploying your code. With PAAS by contrast you can drop down to the command line if absolutely necessary, but the platform encourages you to use higher order abstractions instead. The advantage is that this can save you time; the disadvantage is that if you want to do something that the platform authors didn't expect, you need to get back to the hardware layer (and that may not always be feasible). Finally, with SAAS you are interacting solely with an API or GUI and have zero control over the hardware.

Linux Basics

We now have a bit more vocabulary to go through in slow motion what we did during the previous lecture. First, you used an `ssh` client on your local Windows or Mac to connect to an AWS instance, a virtual machine running Ubuntu 12.04.2 LTS in one of their datacenters. Upon connecting to the instance, you were presented with a `bash` shell that was waiting for our typed-in commands, much like the Google Search prompt waits for your typed-in searches.



The screenshot shows a terminal window titled "ubuntu@ip-172-31-1-161: ~ — ssh — 156x40". The terminal content is as follows:

```
[balajis@junit:~]$ cd downloads
[balajis@junit:~/downloads]$ chmod 400 my-ssh-key-pair.pem
[balajis@junit:~/downloads]$ ssh -i my-ssh-key-pair.pem ubuntu@ec2-54-218-35-71.us-west-2.compute.amazonaws.com
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-40-virtual x86_64)

 * Documentation:  https://help.ubuntu.com/
 
System load:  0.0          Processes:      58
Usage of /:   14.4% of 7.87GB  Users logged in:  0
Memory usage: 42%           IP address for eth0: 172.31.1.161
Swap usage:   0%
 
Graph this data and manage this system at https://landscape.canonical.com/
 
Get cloud support with Ubuntu Advantage Cloud Guest:
  http://www.ubuntu.com/business/services/cloud
 
Use Juju to deploy your cloud instances and workloads:
  https://juju.ubuntu.com/#cloud-precise

ubuntu@ip-172-31-1-161:~$
```

A red box highlights the command `ssh -i my-ssh-key-pair.pem ubuntu@ec2-54-218-35-71.us-west-2.compute.amazonaws.com`. A red arrow points from this box to a note on the right.

Note that we specify the username (`ubuntu@ec2...`)

Figure 2: `ssh` connection to your EC2 instance, pulling up a `bash` shell.

During the course you will execute many thousands of commands in this environment, so let's first get familiar with the basics of navigating around the filesystem.

Filesystem Basics

You're familiar with the concept of directories and files from working with your local computer. Linux has the same concepts; the main difference is that you generally navigate between directories and create/read/update/delete files by simply typing in commands rather than clicking on files. Let's go through a short interactive example which illustrates how to create and remove files and directories and inspect their properties:

```
1 # -- Executed on remote machine
2 cd $HOME # change to your home directory
3 pwd # print working directory
4 mkdir mydir # make a new directory, mydir
5 cd mydir
6 pwd # now you are in ~/mydir
7 touch myfile # create a blank file called myfile
8 ls myfile
9 ls -alrth myfile # list metadata on myfile
10 alias ll='ls -alrth' # set up an alias to save typing
11 ll myfile
12 echo "line1" >> myfile # append via '>>' to a file
13 cat myfile
14 echo "line2" >> myfile
15 cat myfile
16 cd ..
17 pwd
18 cp mydir/myfile myfile2 # copy file into a new file
19 cat myfile2
20 cat mydir/myfile
21 ls -alrth myfile2 mydir/myfile
22 rm -i myfile2
23 cp -av mydir newdir # -av flag 'archives' the directory, copying timestamps
24 rmdir mydir # won't work because there's a file in there
25 rm -rf mydir # VERY dangerous command, use with caution
26 cd newdir
27 pwd
28 cp myfile myfile-copy
29 echo "line3" >> myfile
30 echo "line4" >> myfile-copy
31 mv myfile myfile-renamed # mv doubles as a rename
32 ll
33 cat myfile-renamed
34 cat myfile-copy
35 ll
36 rm myfile-*
37 ll
```

```
38 cd ..
39 ll
40 rmdir newdir
41 ll
```

This should give you an intuitive understanding of how to navigate between directories (`cd`), print the current working directory (`pwd`), print the contents of files (`cat`), list the contents of directories (`ls`), copy files (`cp`), rename files (`mv`), move files (`mv again`), and remove files and directories (`rm`). We'll do much more on this, but you can read more [here](#), [here](#), and [here](#). You might now execute `ls -alrth /` and use `cd` to explore the [Ubuntu directory hierarchy](#).

env: List all environmental variables

Just like you can configure parameters like “number of results” which affect what Google Search does as you type in commands, so too can you configure parameters for `bash` called *environmental variables* that will produce different behavior for the same commands. You can see all environmental variables defined in `bash` by running the¹ command `env`:

```
1 # -- Executed on remote machine
2 $ env
```

There's an example in Figure 3. We can see for example the current value of `SHELL` is `/bin/bash`, the current value of `USER` is `ubuntu`, and so on. But perhaps the most important is the `PATH`.

¹`env` can also be used to set up temporary environments for other commands, or in the [sha-bang](#) line of a shell script. See [here](#) for more.

```
ubuntu@ip-172-31-1-161:~$ env
TERM=xterm-256color
SHELL=/bin/bash
SSH_CLIENT=166.230.148.156 61786 22
SSH_TTY=/dev/pts/0
LC_ALL=POSIX
USER=ubuntu
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
01:or=40;31:01:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;
31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.lzma=01;31:*.lzo=01;
*:zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lz=01;31:*.bz=01;
31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.war=01;
31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.rz=01;
31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.ppm=01;35:*.tga=01;
35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;
35:*.pcx=01;35:*.mov=01;35:*.n=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;
35:*.ogm=01;35:*.mp4=01;35:*.m4=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;
35:*.wmv=01;35:*.ASF=01;35:*.rm=01;35:*.c=01;35:*.avi=01;35:*.fli=01;
35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;
35:*.axv=01;35:*.anx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.mid=00;
36:*.midi=00;36:*.mkcp=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.axa=00;
36:*.oga=00;36:*.oga=00;36:*.oga=00;36:*.oga=00;36:*.oga=00;36:*.oga=00;
HOME=/home/ubuntu
LOGNAME=ubuntu
SSH_CONNECTION=166.230.148.156 61786 172.31.1.161 22
LESSOPEN=- /usr/bin/lesspipe %s
LESSCLOSE=/usr/bin/lesspipe %s %s
_=~/usr/bin/env
```

Figure 3: The `env` command prints the bash environment variables.

PATH: Where Linux looks for programs

Perhaps the most important environmental variable is the PATH. It defines the order of directories which the computer will search for programs to execute. This is important when you have installed a program in a non-standard place, or when multiple versions of a program with the same name exist. For example, suppose you have two different versions of git installed on your machine in /usr/bin/git and /usr/local/bin/git. If you don't specify the full pathname on the command line and simply type in git, the OS uses the PATH to determine which one you mean. Here are two ways to see² the current value of the PATH:

```
1 # -- Executed on remote machine
2 $ env | grep '^PATH'
3 $ echo $PATH
```

We see in the case of our stock Ubuntu EC2 12.04.2 LTS instance that the PATH's value is:

```
1 # -- Executed on remote machine
2 $ echo $PATH
3 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

This is a colon-delimited list of directory names. It means the OS will search directories for programs in the following precedence order:

- /usr/local/sbin
- /usr/local/bin
- /usr/sbin
- /usr/bin
- /sbin
- /bin
- /usr/games

You can see [this post](#) for more on why this particular order is the default search path for Ubuntu. But the upshot for our purposes is that if /usr/local/bin/git and /usr/bin/git both existed, then /usr/local/bin/git would take precedence because the former directory (/usr/local/bin) came before the latter (/usr/bin) in the PATH (/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games). See [here](#) for more.

²Note that the reason we prepend a \$ to the beginning of PATH is that this is how the bash shell scripting language identifies variable names (see [here](#) for more).

HOME: The current user's home directory

The value of another environmental variable also deserves a brief mention. `HOME` is also known as `~` or ‘tilde’ or the home directory; if you execute `echo $HOME` at the prompt, you will see `/home/ubuntu`, as we are currently logged in as the user named `ubuntu`. You can change to the home directory at any time by typing `cd $HOME`, `cd ~`, or just `cd` and then hitting enter. You can also use `HOME` to make your scripts more portable, rather than [hardcoding](#) something like `/home/ubuntu`. By convention, files that are specific to a given user are placed in the home directory. See [here](#) for more.

which: How to debug PATH issues

In some ways the `PATH` is just a convenience, as in theory you could type in the exact full address of every resource on the computer, but local nicknames save a lot of typing. However, `PATH` bugs can be very confusing for newcomers to the command line, and are often the source of subtle bugs³ even for experienced engineers. The most frequent issue is when you *thought* you installed, upgraded, or recompiled a program, but the `PATH` pulls up an older version or doesn't include your new program. The way to debug this is with the `which` command, as it will tell you where the program you are invoking is located. Try executing the following series of commands:

```
1 # -- Executed on remote machine
2 $ which git
3 $ sudo apt-get install -y git-core
4 $ which git
```

The first `which git` command may print nothing, if you haven't installed `git` via `apt-get`. The second will print `/usr/bin/git` for certain because you've just installed it. Moreover, you can try executing the following:

```
1 # -- Executed on remote machine
2 $ /usr/bin/git --help
3 $ git --help
```

Why do both of those invocations work? Because `/usr/bin` is in the `PATH`, you can just type in `git` and it will be found. If `git` had been installed in a different location, outside of the `PATH` directories, the second command wouldn't work. So `which` can be used as a basic debugging step to just confirm that the program you are trying to run is even findable in the `PATH`.

What gets more interesting is when there are multiple versions of the same program. For illustrative purposes, let's introduce such a conflict by executing the following commands:

```
1 # -- Executed on remote machine
2 $ which -a git
```

³This is especially true when compiling C code, as there is not just one `PATH` variable but [several such variables](#) for the locations of linker files and the like.

```

3 /usr/bin/git
4 $ git
5 usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
6 ... [snipped]
7 $ cd $HOME
8 $ mkdir bin
9 $ wget https://spark-public.s3.amazonaws.com/startup/code/git
10 $ mv git bin/
11 $ chmod 777 bin/git
12 $ echo $PATH
13 /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
14 $ export PATH=$HOME/bin:$PATH
15 $ echo $PATH
16 /home/ubuntu/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
17 $ which -a git
18 /home/ubuntu/bin/git
19 /usr/bin/git
20 $ git
21 fake git
22 $ /home/ubuntu/bin/git
fake git
24 $ /usr/bin/git
25 usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]
27 $ export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
28 $ git
29 usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
30 ... [snipped]
31 $ /home/ubuntu/bin/git
fake git
33 $ /usr/bin/git
usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]

```

In this example, we first confirmed that the normal `git` was found in the path (after installation). We then downloaded a second file also named `git`, put it in `~/bin`, and made it executable⁴ with `chmod 777`. We next used the `export` command to redefine the `$PATH` variable, adding `/$HOME/bin` to the front of the path. Then, when we ran `which -a git`, a new entry came to the top: `/home/ubuntu/bin/git`, which is our fake command. And now typing in `git` at the command line just prints out our dummy string, because the precedence order in the `PATH` was changed and `/home/ubuntu/bin/git` came first. Last, we restore the value of the `PATH` and now invoke `git` again. Note finally that no matter what the value of the `PATH`, giving absolute paths to `/home/ubuntu/bin/git` and `/usr/bin/git` always works to specify exactly which `git` you want to execute; the `PATH` is for resolving the shortcut of just typing `git`.

⁴In general, one should be cautious about executing code downloaded from the internet, but in this case this file is from a trusted URL.

ssh: Connect to remote machines

The `ssh` command allows you to [securely connect](#) to a remote machine. It was a replacement for the old `telnet` command, which sent your password in the clear. Released in 1995, `ssh` really ramped up after the Internet became reasonably popular and [security](#) became a concern. Here's some `ssh` commands similar⁵ to what we saw in the last lecture:

```
1 # -- Executed on local machine
2 # Login to AWS as the ubuntu user using your private key pair
3 $ cd downloads # or wherever your pem file is
4 $ ssh -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com
5
6 # Login to AWS as ubuntu using your private key pair and run uptime command
7 $ ssh -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com uptime
```

Passing all that information on the command line each time is a little unwieldy, so one way to shorten it is to put some of the information into the `~/.ssh/config` file, as detailed [here](#). When you are done, your config file should look something like this when you `cat` the file⁶:

```
1 # -- Execute on local machine
2 $ mkdir -p ~/.ssh
3 $ cp ~/downloads/skey.pem ~/.ssh/
4 $ chmod 400 ~/.ssh/skey.pem
5 $ chmod 700 ~/.ssh
6 $ nano ~/.ssh/config # edit the file as shown below
7 $ cat ~/.ssh/config
8 Host awshost1
9 HostName ec2-54-218-35-71.us-west-2.compute.amazonaws.com
10 User ubuntu
11 IdentityFile "~/ssh/skey.pem"
```

And then you can simply do this:

```
1 # -- Execute on local machine
2 # SSH using the 'awshost1' alias defined in ~/.ssh/config
3 $ ssh awshost1
4
5 # SSH using an alias, run a command, and then exit
6 $ ssh awshost1 uptime
```

⁵Note that you will need to use your own `pem` files and hostname in order to make this work. You will need to be in the directory with the `.pem` file (probably `~/Downloads` on a Mac) in order to pass it in as a command line argument with `-i..`.

⁶In this context, `cat` the file means to print the file to the screen.

The one disadvantage here is that you will need to reenter this information every time you instantiate a new EC2 instance. However, next time⁷ you can [stop rather than terminate](#) your EC2 instance when you disconnect. This won't guarantee that the host is still up (Amazon can in theory terminate your instances at any time), but it makes it more likely that you won't have to constantly edit this config, and might actually ensure you stay under the free tier hours ([see here](#)).

In conclusion, these few commands are most of what you need to know about `ssh`, but as a supplement, here are some further `ssh` [examples](#), and some even more [sophisticated things](#) with `ssh` (useful once we learn about [pipes](#)).

`scp`: Copy files to/from remote machines.

Just like `ssh` lets you connect to a remote machine⁸ and run commands on it, you can use the allied `scp` command to copy files back and forth from the remote machine. This will be very useful for doing homework, as you can edit/generate files on the remote computer and bring them to the local machine, or download datasets for homework assignments through your web browser and then upload them to the remote machine. Here are a few `scp` examples:

```

1 # -- Execute on local computer
2 # Copy hello.txt from local computer to remote home directory
3 $ scp -i skey.pem hello.txt ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/
4
5 # Copy hello.txt from local to remote home directory, renaming it foo.txt
6 $ scp -i skey.pem hello.txt ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt .
7
8 # Copying ~/foo.txt from remote to current local directory
9 $ scp -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt .
10
11 # Copying ~/foo.txt from remote to local directory cc, renaming it a.b
12 $ scp -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt cc/a.b

```

All of these become simpler when you set up your `.ssh/config` file (see this post and the previous section on `ssh`). Then you can do:

```

1 # -- Execute on local computer
2 # Copy hello.txt from local computer to remote home directory
3 $ scp hello.txt awshost1:~/
4
5 # Copy hello.txt from local to remote home directory, renaming it foo.txt
6 $ scp hello.txt awshost1:~/foo.txt

```

⁷One major caveat: given the scale of the class, we might actually want to terminate rather than stop instances to ensure that Amazon has enough capacity. Even for Amazon, 100,000 virtual machines is potentially a big number. They are giving us a free product, so we should be good hosts and not inadvertently abuse that generosity.

⁸Note that it is fairly common to get confused about which box you are currently running commands on, which is why it is common to run the `whoami`, `uname`, or `hostname` commands (or put them in your `$PROMPT` as we will see) to determine which machine you're on. Try them out: they will return different results on your local machine and on your remote EC2 instance.

```

7
8 # Copying ~/foo.txt from the remote computer to the current local directory
9 $ scp awshost1:~/foo.txt .
10
11 # Copying ~/foo.txt from remote to local directory cc, renaming it a.b
12 $ scp awshost1:~/foo.txt cc/a.b

```

Note: all of these `scp` invocations will overwrite the target file if it already exists (aka **clobbering**), but they will not create a directory if it does not already exist. For that you need the **more powerful** `rsync` command, which we will cover later. However, you now have the commands necessary to copy homework-related files to and from your local computer to your running EC2 instance.

`bash`: Command interpreter

A command line interface (CLI) is to a search engine as `bash` is to Google. That is, the CLI is the abstract concept and `bash` is one of several specific instantiations called **shells**. Just like there are many search engines, there are many shells: `tcsh`, `ksh`, `bash`, `zsh`, and more. `zsh` is **superior** to (and reverse-compatible with) `bash` in many ways, but what `bash` has going for it, especially in an introductory class, is ubiquity and standardization. One of the important points about `bash` is that it both has its own built-in commands AND is used to invoke user-installed commands. This is roughly analogous to the way that Google can present both its own results (like a calculator or flight results) or route you to blue links leading to other sites when you type in a command. Here are a few commands to try:

```

1 # -- Execute on EC2 instance
2 # Print all bash environmental variables
3 $ env
4 # Show the value of the HOME variable
5 $ echo $HOME
6 # Change directories to HOME
7 $ cd $HOME
8 # Run a few different ls commands
9 $ ls
10 $ ls *
11 $ ls -alrth *

```

Given how much time you will spend using `bash`, it's important to learn all the **keyboard shortcuts**. One of the important things about `bash` is that often you will want to execute many commands in a row. You can do this by putting those commands into a single file called a **shell script**. Often an end user will want some logic in these commands, e.g. to install a package if and only if the operating system is above a particular version. And so it turns out that `bash` actually includes a full fledged programming language, which is often used to script installs (here's a **sophisticated example**). We'll use something similar to rapidly configure a new EC2 instance later in the class, but for now here's an example of a simple **shell script**:

```
1 #!/bin/bash
2 date
3 echo "Good morning, world"
```

The line at the top is called the sha-bang ([more here](#)) and is a bit of magic that tells the shell what program to use⁹ to interpret the rest of the script.

Unlike the previous code, we don't type the above directly into the prompt. Instead we need to save it as a file. Below are some commands that will accomplish that task; just type them in now, you'll understand more later.

```
1 # -- Execute on EC2 instance
2 # Download a remote file
3 $ wget https://spark-public.s3.amazonaws.com/startup/code/simple.sh
4 # Print it to confirm you have it; should show a file with "Good morning, world"
5 $ cat simple.sh
6 # Check permissions
7 $ ls -alrth simple.sh
8 # Set to executable
9 $ chmod 777 simple.sh
10 # Confirm that permissions are changed
11 $ ls -alrth simple.sh
12 # Execute the file from the current directory
13 $ ./simple.sh
```

That simple script should print the date (Figure 4).

⁹For example, in this case it's using `/bin/bash`. If instead we typed in `/usr/bin/env node`, it would run the `env` command to find the `node` binary and use that to interpret the rest of the file; this is useful when you might have [multiple versions](#) of the node binary.

```

ubuntu@ip-172-31-1-161:~$ wget https://spark-public.s3.amazonaws.com/startup/code/simple.sh
--2013-06-21 11:26:25-- https://spark-public.s3.amazonaws.com/startup/code/simple.sh
Resolving spark-public.s3.amazonaws.com (spark-public.s3.amazonaws.com)... 72.21.195.15
Connecting to spark-public.s3.amazonaws.com (spark-public.s3.amazonaws.com)|72.21.195.15|:443
HTTP request sent, awaiting response... 200 OK
Length: 44 [application/octet-stream]
Saving to: `simple.sh'

100%[=====] 2013-06-21 11:26:25 (10.2 MB/s) - `simple.sh' saved [44/44]

ubuntu@ip-172-31-1-161:~$ cat simple.sh
#!/bin/bash
date
echo "Good morning, world"
ubuntu@ip-172-31-1-161:~$ ls -alrth simple.sh
-rw-rw-r-- 1 ubuntu ubuntu 44 Jun 21 11:21 simple.sh
ubuntu@ip-172-31-1-161:~$ chmod 777 simple.sh
ubuntu@ip-172-31-1-161:~$ ls -alrth simple.sh
-rwxrwxrwx 1 ubuntu ubuntu 44 Jun 21 11:21 simple.sh
ubuntu@ip-172-31-1-161:~$ ./simple.sh
Fri Jun 21 11:26:44 UTC 2013
Good morning, world
ubuntu@ip-172-31-1-161:~$ 
```

1. Download

2. Make executable

3. Execute

Figure 4: Simple script that prints the date

To recap, we just covered `bash` basics: entering commands, `bash` keyboard shortcuts, and the idea of shell scripts. `bash` actually has surprising depth, and you can see more by typing in `man bash` (press q to quit); but let's move on for now.

apt-get: Binary package management

There are two major ways to install software in Linux: from *binary packages* and *from source*. Binary packages are pre-compiled packages for your computer's architecture that can be installed as rapidly as they are downloaded. Here is a quick example of installing `git` via Ubuntu's binary package manager, `apt-get`:

```
1 # -- Execute on remote machine
2 $ which git
3 $ sudo apt-get install -y git-core
4 # ... lots of output ...
5 $ which git
6 /usr/bin/git
```

That was fast and easy (and you can learn more `apt-get` commands [here](#)). There are disadvantages to binary packages, though, including:

- because they are pre-compiled, you will not be able to customize binaries with compile-time flags that change behavior in fundamental ways, which is especially important for programs like `nginx`.
- only older versions of packages will usually be available in binary form.
- newer or obscure software may not be available at all in binary form.
- they may be a bit slower or not 100% optimized for your architecture, especially if your architecture is odd or custom; often this is related to the compile-time flag issue.

That said, for most non-critical software you will want to use binary packages. We'll cover installation from source later.

Linux Recap and Summary

All right. So now we can understand how to initialize a virtual machine, connect to it via ssh, launch a `bash` shell, and install some software via binary packages to configure our machine. We also can move around and create files, and perhaps know a bit about the Ubuntu directory hierarchy. When combined with the previous lectures, we can now dial up a Linux machine in the cloud, configure it to a fair extent, and start executing code! Awesome progress. Now let's write a little code.

Basic Server-Side JS (SSJS)

Install node and npm

We're going to start in this first lesson by treating `node` as just another scripting language like `python` or `perl`, without any of the special features of `node` (networking, async invocation, shared client-server code). Let's begin by executing the following commands on an EC2 instance to set up `node` and `npm` (the `node` package manager).

```
1 $ sudo apt-get update
2 # Install a special package
3 $ sudo apt-get install -y python-software-properties python g++ make
4 # Add a new repository for apt-get to search
5 $ sudo add-apt-repository ppa:chris-lea/node.js
6 # Update apt-get's knowledge of which packages are where
7 $ sudo apt-get update
8 # Now install nodejs and npm
9 $ sudo apt-get install -y nodejs
```

And now you have the `node` and `npm` commands. You can check that these exist by invoking them as follows:

```
1 $ npm --version
2 1.2.32
3 $ node --version
4 v0.10.12
```

The `node.js` REPL

A REPL is a read-evaluate-print loop. It refers to an environment where you can type in commands in a programming language, one line at a time, and then immediately see them execute. You can launch the `node` REPL at the prompt by simply typing `node`:

```
[ubuntu@ip-10-204-245-82:~]$node
> 1 + 1
2
> var foo = "asdf";
undefined
> foo
'asdf'
> .help
.break Sometimes you get stuck, this gets you out
.clear Alias for .break
.exit Exit the repl
.help Show repl options
.load Load JS from a file into the REPL session
.save Save all evaluated commands in this REPL session to a file
> █
```

Figure 5: The node.js Read-Evaluate-Print-Loop (REPL) is launched by simply typing `node` at the command line after installing it.

Editing code with nano

We'll use the `nano` code editor for your first few programs; using something primitive/simple like this will increase your appreciation for the combination of `emacs` and `screen`. As shown in Figure 6, it is useful to open up two separate connections to your remote machine (e.g. two Terminal.app windows on a Mac, or two Cygwin windows on Windows):

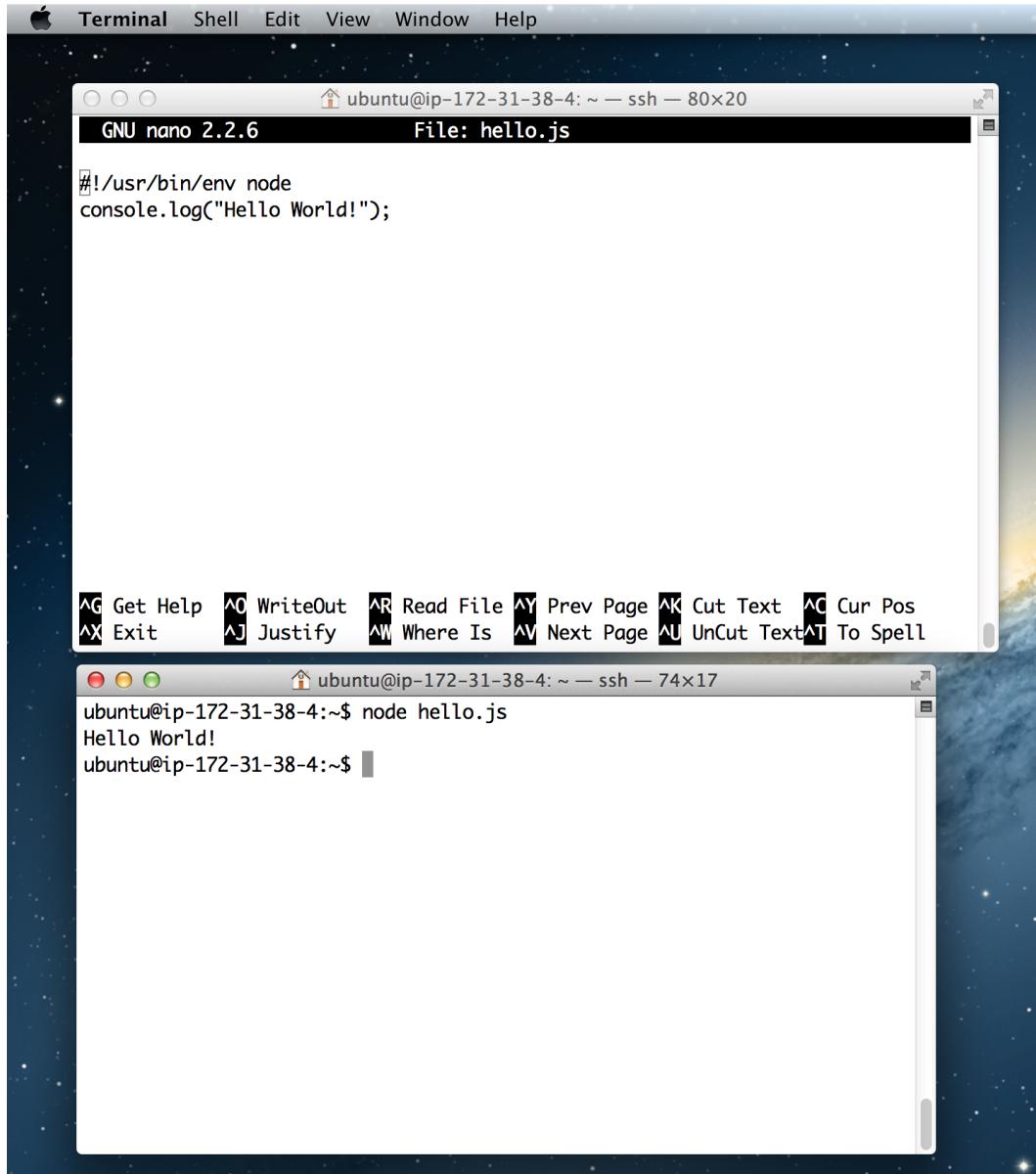


Figure 6: Here's an example of editing with two windows open and connected to the remote host, one for the `nano` editor and one with the main `bash` prompt. We'll see a much better way to do this soon with `GNU screen`.

A first node.js program: Hello World

If you want to zip ahead with node, you can't go wrong by going through the [Node.js Beginner's Book](#). But let's do our first node.js program, just to confirm it works. Open a file named `hello.js` with the `nano` editor by typing `nano hello.js`. Type in `#!/usr/bin/env node`, hit enter, type in `console.log("Hello World");` and then save and quit (by hitting Control-X, then yes, then Enter). Then execute the file with `node hello.js`. Here is how that should look:

```
1 [ubuntu@ip-10-204-245-82:~/nodedemo]$nano hello.js # edit as described above
2 [ubuntu@ip-10-204-245-82:~/nodedemo]$cat hello.js
3 #!/usr/bin/env node
4 console.log("Hello World");
5 [ubuntu@ip-10-204-245-82:~/nodedemo]$node hello.js
6 Hello World
```

If this works, great job. You've got a working interpreter and can do the first part of the [HW1 programming assignment](#).

A second node.js program: Using the libraries

Now try typing in the following script with `nano` in a file named `hello2.js`:

```
1 #!/usr/bin/env node
2 var fs = require('fs');
3 var outfile = "hello.txt";
4 var out = "Modify this script to write out something different.\n";
5 fs.writeFileSync(outfile, out);
6 console.log("Script: " + __filename + "\nWrote: " + out + "To: " + outfile);
```

The two new wrinkles of note are that we are using the `writeFileSync` function from the built-in filesystem (`fs`) library ([more details](#) if you want, but don't worry about them yet) and are also using the built-in node.js convention that `__filename` is a variable with the path to the current file. Here is what it looks like if we print the file to the screen with `cat` and then execute it:

```
1 ubuntu@ip-172-31-1-161:~$ cat hello2.js
2 #!/usr/bin/env node
3 var fs = require('fs');
4 var outfile = "hello.txt";
5 var out = "Modify this script to write out something different.\n";
6 fs.writeFileSync(outfile, out);
7 console.log("Script: " + __filename + "\nWrote: " + out + "To: " + outfile);
8
9 ubuntu@ip-172-31-1-161:~$ node hello2.js
10 Script: /home/ubuntu/hello2.js
11 Wrote: A startup is a business built to grow rapidly.
```

```

12 To: hello.txt
13
14 ubuntu@ip-172-31-1-161:~$ cat hello.txt
15 A startup is a business built to grow rapidly.

```

As shown above, if you `cat hello.txt` it has the string we expected. Note also that you can use `chmod` to make the `hello2.js` script executable, as follows:

```

1 ubuntu@ip-172-31-1-161:~$ chmod 777 hello2.js
2
3 ubuntu@ip-172-31-1-161:~$ ./hello2.js
4 Script: /home/ubuntu/hello2.js
5 Wrote: Modify this script to write out something different.
6 To: hello.txt

```

In this example `bash` is using the [sha-bang](#) `#!/usr/bin/env node` in the first line of `hello2.js` to determine what program to use to interpret the script. Don't worry about it if you don't get it fully yet - for now, just remember to include that sha-bang line (`#!/usr/bin/env node`) at the top of any `node` server-side script, and then do `chmod 777` to make it executable.

A third node.js program: Fibonacci

Here's our third node program. Do `nano fibonacci.js` and type in¹⁰ the following code. You can of course copy it locally and `scp` it to the remote machine, but often [typing in code yourself](#) is a good exercise.

```

1 #!/usr/bin/env node
2
3 // Fibonacci
4 // http://en.wikipedia.org/wiki/Fibonacci_number
5 var fibonacci = function(n) {
6     if(n < 1) { return 0;}
7     else if(n == 1 || n == 2) { return 1;}
8     else if(n > 2) { return fibonacci(n - 1) + fibonacci(n - 2);}
9 };
10
11 // Fibonacci: closed form expression
12 // http://en.wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence
13 var fibonacci2 = function(n) {
14     var phi = (1 + Math.sqrt(5))/2;
15     return Math.round((Math.pow(phi, n) - Math.pow(1-phi, n))/Math.sqrt(5));
16 };
17
18 // Find first K Fibonacci numbers via basic for loop

```

¹⁰OK, if you are really lazy, just do this: `wget https://spark-public.s3.amazonaws.com/startup/code/fibonacci.js`

```

19 var firstkfib = function(k) {
20     var i = 1;
21     var arr = [];
22     for(i = 1; i < k+1; i++) {
23         arr.push(fibonacci(i));
24     }
25     return arr;
26 };
27
28 // Print to console
29 var fmt = function(arr) {
30     return arr.join(" ");
31 };
32
33 var k = 20;
34 console.log("firstkfib(" + k + ")");
35 console.log(fmt(firstkfib(k)));

```

In general, if you have a CS106B level background as listed in the [prerequisites](#), this should be a fairly straightforward bit of code to understand. It uses basic programming constructs like variables, if/else statements, for loops, functions, and the like.

The only things that might not be completely obvious are the use of the `push` and `join` methods from the `Array` object ([see here](#)), and the use of the `round`, `pow`, and `sqrt` methods from the `Math` object ([see here](#)). The function declaration syntax might look a little weird too (in the sense of assigning a function to a named variable), but you can just accept it for now as a preview of what is to come ([1](#), [2](#), [3](#)). Executing this script should give you the first 20 Fibonacci numbers:

```

1 [balajis@jiunit:~]$chmod 777 fibonacci.js
2 [balajis@jiunit:~]$./fibonacci.js
3 firstkfib(20)
4 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

```

Once you understand this short script, you should be able to adapt it to solve the prime number calculation problem in the homework.

The Linux Command Line

In this lecture we'll introduce you to a large number of new command line tools and show you how to connect them together via pipes.

The Command Line

In general, the best way to understand the command line is to learn by doing. We'll go through a number of useful commands with examples, and you should become reasonably skilled in the process, but it is nevertheless worth looking at a few books to consult as references. Here are some particularly good ones:

- [The Command Line Crash Course](#) is Zed Shaw's free online tutorial introduction to the command line. Much shorter than the others listed here and recommended.
- [Sobell's Linux Book](#): If you only get one book, get Sobell's. It is probably the single best overview, and much of the content is applicable to both Linux and OS X¹ systems.
- [Unix Power Tools](#): While this came out in 2002, much of it is still relevant to the present day. New options have been added to commands but old ones are very rarely deprecated.
- [Unix/Linux Sysadmin Handbook](#): Somewhat different focus than Sobell. Organized by purpose (e.g. shutdown), and a good starter guide for anyone who needs to administer machines. Will be handy if you want to do anything nontrivial with EC2.

With those as references, let's dive right in. As always, the commands below should be executed after `ssh`'ing into your EC2 virtual machine¹. If your machine is messed up for some reason, feel free to terminate the instance in the [AWS EC2 Dashboard](#) and boot up a new one. Right now we are doing all these commands in a “vanilla” environment without much in the way of user configuration; as we progress we will set up quite a lot.

The three streams: STDIN, STDOUT, STDERR

Most `bash` commands [accept input](#) as a single stream of bytes called `STDIN` or standard input and yields two output streams of bytes: `STDOUT`, which is the standard output, and `STDERR` which is the stream for errors.

- `STDIN`: this can be text or binary data streaming into the program, or keyboard input.
- `STDOUT`: this is the stream where the program writes its data. This is printed to the screen unless otherwise specified.

¹You can probably get away with running them on your local Mac, but beware: the built-in command line tools on OS X (like `mv`) are very old BSD versions. You want to install the new GNU tools using [homebrew](#). Then your local Mac environment will be very similar to your remote EC2 environment.

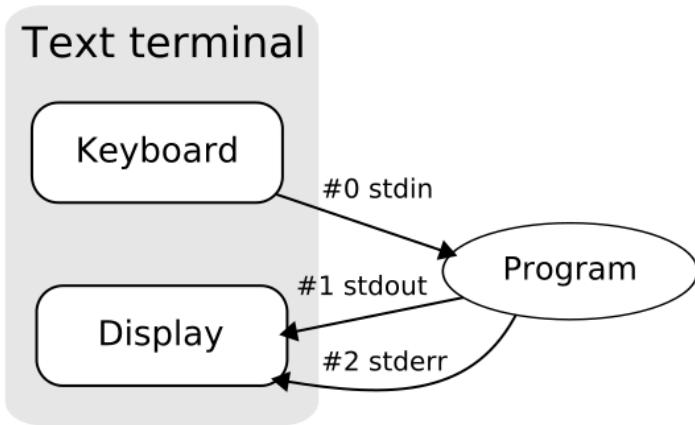


Figure 1: Visualizing the three standard streams. From [Wikipedia](#).

- **STDERR:** this is the stream where error messages are displayed. This is also printed to the screen unless otherwise specified.

Here are a few simple examples; execute these at your EC2 command line.

```

1 # Redirecting STDOUT with >
2 echo -e "line1\nline2"
3 echo -e "line1\nline2" > demo.txt
4
5 # Redirecting STDERR with 2>
6 curl fakeurl # print error message to screen
7 curl fakeurl 2> errs.txt
8 cat errs.txt
9
10 # Redirecting both STDIN and STDOUT to different files with 1> and 2>
11 curl google.com fakeurl 1> out1.txt 2> out2.txt
12
13 # Redirecting both to the same file with &>
14 curl google.com fakeurl &> out.txt
15
16 # Getting STDIN from a pipe
17 cat errs.txt | head
18
19 # Putting it all together
20 curl -s http://google.com | head -n 2 &> asdf.txt

```

Note what we did in the last two examples, with the `|` symbol. That's called a pipe, and the technique of connecting one command's `STDOUT` to another command's `STDIN` is very powerful. It allows us to build up short-but-powerful programs by composing individual pieces. If you *can* write something this way, especially for text processing or data analysis, you almost always *should*...even if it's a complex ten step pipeline. The reason is that it will be incredibly fast and robust, due to the fact that the underlying GNU tools are written in C

and have received countless hours of optimization and bug fixes. It will also not be that hard to understand: it's still a one-liner, albeit a long one. See this piece on [Taco Bell programming](#) and [commentary](#). As a corollary, you should design your own programs to work in command line pipelines (we'll see how to do this as the course progresses). With these principles in mind, let's go through some interactive examples of Unix commands. All of these should work on your default Ubuntu EC2 box.

Navigation and Filesystem

list files: ls

The most basic and heavily used command. You will use this constantly to get your bearings and confirm what things do.

```
1 ls
2 ls --help
3 ls --help | less # 'pipe' the output of ls into less
4 ls -a
5 ls -alrth
6 alias ll='ls -alrth' # create an alias in bash
7 ll
```

modify/create empty files: touch

Used to change timestamps and quickly create zero byte files as placeholders. Useful for various kinds of tests.

```
1 touch file1
2 touch file2
3 ll
4 ll --full-time
5 touch file1
6 ll --full-time # note that the file1 modification time is updated
```

display text: echo

Useful for debugging and creating small files for tests.

```
1 echo "foo"
2 man echo
3 echo -e "foo\n"
4 echo -e "line1\nline2" > demo.txt # write directly to tempfile with '>'
```

copy files: cp

Powerful command with many options, works on everything from single files to entire archival backups of huge directories.

```

1 cp --help
2 cp demo.txt demo2.txt
3 cp -v demo.txt demo3.txt # verbose copy
4 cp -a demo.txt demo-archive.txt # archival exact copy
5 ll --full-time demo* # note timestamps of demo/demo-archive
6 echo "a new file" > asdf.txt
7 cat demo.txt
8 cp asdf.txt demo.txt
9 cat demo.txt # cp just clobbered the file
10 alias cp='cp -i' # set cp to be interactive by default

```

move/rename files: mv

Move or rename files. Also extremely powerful.

```

1 mv asdf.txt asdf-new.txt
2 mv -i asdf-new.txt demo.txt # prompt before clobbering
3 alias mv='mv -i'

```

remove files: rm

Powerful command that can be used to delete individual files or recursively delete entire trees. Use with caution.

```

1 rm demo.txt
2 rm -i demo2.txt
3 rm -i demo*txt
4 alias rm='rm -i'
5 alias # print all aliases

```

symbolic link: ln

Very useful command that allows a file to be symbolically linked, and therefore in two places at once. Extremely useful for large files, or for putting in one level of indirection.

```

1 yes | nl | head -1000 > data1.txt
2 head data1.txt
3 ll data1.txt
4 ln -s data1.txt latest.txt
5 head latest.txt
6 ll latest.txt # note the arrow. A symbolic link is a pointer.
7 yes | nl | head -2000 | tail -50 > data2.txt
8 head latest.txt
9 ln -s data2.txt latest.txt # update the pointer w/o changing the underlying file
10 head latest.txt

```

```
11 head data1.txt  
12 head data2.txt
```

print working directory: `pwd`

Simple command to print current working directory. Used for orientation when you get lost.

```
1 pwd
```

create directories: `mkdir`

Create directories. Has a few useful options.

```
1 mkdir dir1  
2 mkdir dir2/subdir      # error  
3 mkdir -p dir2/subdir # ok
```

change current directory: `cd`

Change directories. Along with `ls`, one of the most frequent commands.

```
1 cd ~  
2 cd dir1  
3 pwd  
4 cd ..  
5 cd dir2/subdir  
6 pwd  
7 cd - # jump back  
8 cd - # and forth  
9 cd # home  
10 alias ..='cd ..'  
11 alias ...='cd ..; cd ..'  
12 cd dir2/subdir  
13 ...  
14 pwd
```

remove directories: `rmdir`

Not used that frequently; usually recursive `rm` is used instead. Note that `rm -rf` is the most dangerous command in Unix and must be used with extreme caution, as it means “remove recursively without prompting” and can easily nuke huge amounts of data.

```
1 rmdir dir1  
2 rmdir dir2 # error  
3 rm -rf dir2 # works
```

Downloading and Syncing

synchronize local and remote files: rsync

Transfer files between two machines. Assuming you have set up your `~/.ssh/config` as in previous lectures, run this command on your *local* machine, making the appropriate substitutions:

```
1 # Run on local machine
2 $ yes | nl | head -500 > something.txt
3 $ rsync -avp something.txt awshost4:~
4 building file list ... done
5 something.txt
6
7 sent 4632 bytes received 42 bytes 1869.60 bytes/sec
8 total size is 4500 speedup is 0.96
9 $ rsync -avp something.txt awshost4:~
10 building file list ... done
11
12 sent 86 bytes received 20 bytes 42.40 bytes/sec
13 total size is 4500 speedup is 42.45
```

Notice that the first `rsync` invocation sent 4632 bytes but the second time sent much less data 86 bytes. That is because `rsync` can resume transfers, which is essential for working with large files and makes it generally preferable to `scp`. As an exercise, `ssh` in and run `cat something.txt` on the EC2 machine to see that it was uploaded.

retrieve files via http/https/ftp: wget

Very useful command to rapidly pull down files or webpages. Note: the distinction between `wget` and `rsync` is that the `wget` is generally used for publicly accessible files (accessible via a web browser, perhaps behind a login) while `rsync` is used for private files (usually accessible only via `ssh`).

```
1 wget http://startup-class.s3.amazonaws.com/simple.sh
2 less simple.sh # hit q to quit
3
4 # pull a single HTML page
5 wget https://github.com/joyent/node/wiki/modules
6 less modules
7
8 # recursively download an entire site, waiting 2 seconds between hits.
9 wget -w 2 -r -np -k -p http://www.stanford.edu/class/cs106b
```

interact with single URLs: curl

A bit different from `wget`, but has some overlap in functionality. `curl` is for interacting with single URLs. It doesn't have the spidering/recursive properties that `wget` has, but it supports

a much wider array of protocols. It is very commonly used as the building block for API calls.

```
1 curl https://install.meteor.com | less # an example install file
2 curl -i https://api.github.com/users/defunkt/orgs # a simple API call
3 GHUSER="defunkt"
4 GHVAR="orgs"
5 curl -i https://api.github.com/users/$GHUSER/$GHVAR # with variables
6 GHVAR="repos"
7 curl -i https://api.github.com/users/$GHUSER/$GHVAR # with diff vars
```

send test packets: ping

See if a remote host is up. Very useful for basic health checks or to see if your computer is online. Has a surprisingly large number of options.

```
1 ping google.com # see if you have internet connectivity
2 ping stanford.edu # see if stanford is up
```

Basic Text Processing

view files: less

Paging utility used to view large files. Can scroll both up and down. Use q to quit.

```
1 rsync --help | less
```

print/concatenate files: cat

Industrial strength file viewer. Use this rather than MS Word for looking at large files, or files with weird bytes.

```
1 # Basics
2 echo -e "line1\nline2" > demo.txt
3 cat demo.txt
4 cat demo.txt demo.txt demo.txt > demo2.txt
5 cat demo2.txt
6
7 # Piping
8 yes | head | cat - demo2.txt
9 yes | head | cat demo2.txt -
10
11 # Download chromosome 22 of the human genome
12 wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.gbk.gz
13 gunzip hs_ref_GRCh37.p10_chr22.gbk.gz
14 less hs_ref_GRCh37.p10_chr22.gbk
15 cat hs_ref_GRCh37.p10_chr22.gbk # hit control-c to interrupt
```

first part of files: head

Look at the first few lines of a file (10 by default). Surprisingly useful for debugging and inspecting files.

```
1 head --help
2 head *gbk      # first 10 lines
3 head -50 *gbk  # first 50 lines
4 head -n50 *gbk # equivalent
5 head *txt *gbk # heads of multiple files
6 head -q *txt *gbk # heads of multiple files w/o delimiters
7 head -c50 *gbk # first 50 characters
```

last part of files: tail

Look at the bottom of files; default is last 10 lines. Useful for following logs, debugging, and in combination with head to pull out subsets of files.

```
1 tail --help
2 tail *gbk
3 head *gbk
4 tail -n+3 *gbk | head  # start at third line
5 head -n 1000 *gbk | tail -n 20 # pull out intermediate section
6 # run process in background and follow end of file
7 yes | nl | head -n 10000000 > foo &
8 tail -F foo
```

extract columns: cut

Pull out columns of a file. In combination with head/tail, can pull out arbitrary rectangular subsets of a file. Extremely useful for working with any kind of tabular data (such as data headed for a database).

```
1 wget ftp://ftp.ncbi.nih.gov/genomes/Bacteria/\
2 Escherichia_coli_K_12_substr_W3110_uid161931/NC_007779.ptt
3 head *ptt
4 cut -f2 *ptt | head
5 cut -f2,5 *ptt | head
6 cut -f2,5 *ptt | head -30
7 cut -f1 *ptt | cut -f1 -d'.' | head
8 cut -c1-20 *ptt | head
```

numbering lines: nl

Number lines. Useful for debugging and creating quick datasets. Has many options for formatting as well.

```
1 nl *gbk | tail -1 # determine number of lines in file
2 nl *ptt | tail -2
```

concatenate columns: paste

Paste together data by columns.

```
1 tail -n+3 *ptt | cut -f1 > locs
2 tail -n+3 *ptt | cut -f5 > genes
3 paste genes locs genes | head
```

sort by lines: sort

Industrial strength sorting command. Very powerful standalone and in combination with others.

```
1 sort genes | less      # default sort
2 sort -r genes | less # reverse
3 sort -R genes | less # randomize
4 cut -f2 *ptt | tail -n+4 | head
```

uniquify lines: uniq

Useful command for analyzing any data with repeated elements. Best used in pipelines with sort beforehand.

```
1 cut -f2 *ptt | tail -n+4 | sort | uniq -c | sort -k1 -rn #
2 cut -f3 *ptt | tail -n+4 | uniq -c | sort -k2 -rn # output number
3 cut -f9 *ptt > products
4 sort products | uniq -d
```

line, word, character count: wc

Determine file sizes. Useful for debugging and confirmation, faster than `nl` if no intermediate information is needed.

```
1 wc *ptt      # lines, words, bytes
2 wc -l *ptt # only number of lines
3 wc -L *ptt # longest line length, useful for apps like style checking
```

split large files: split

Split large file into pieces. Useful as initial step before many parallel computing jobs.

```
1 split -d -l 1000 *ptt subset.ptt.  
2 ll subset.ptt*
```

Help

manuals: man

Single page manual files. Fairly useful once you know the basics. But Google or StackOverflow are often more helpful these days if you are really stuck.

```
1 man bash  
2 man ls  
3 man man
```

info: info

A bit more detail than `man`, for some applications.

```
1 info cut  
2 info info
```

System and Program information

computer information: uname

Quick diagnostics. Useful for install scripts.

```
1 uname -a
```

current computer name: hostname

Determine name of current machine. Useful for parallel computing, install scripts, config files.

```
1 hostname
```

current user: whoami

Show current user. Useful when using many different machines and for install scripts.

```
1 whoami
```

current processes: ps

List current processes. Usually a prelude to killing a process.

```
1 sleep 60 &
2 ps xw | grep sleep
3 # kill the process number
```

Here is how that looks in a session:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
2 [1] 5483
3 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep
4 5483 pts/0 S 0:00 sleep 20
5 5485 pts/0 S+ 0:00 grep --color=auto sleep
6 ubuntu@domU-12-31-39-16-1C-96:~$ kill 5483
7 [1]+ Terminated sleep 20
```

So you see that the process number 5483 was identified by printing the list of running processes with `ps xw`, finding the process ID for `sleep 20` via `grep`, and then running `kill 5483`. Note that the `grep` command itself showed up; that is a common occurrence and can be dealt with as follows:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
2 [1] 5486
3 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep | grep -v "grep"
4 5486 pts/0 S 0:00 sleep 20
5 ubuntu@domU-12-31-39-16-1C-96:~$ kill 5486
```

Here, we used the `grep -v` flag to exclude any lines that contained the string `grep`. That will have some false positives (e.g. it will exclude a process named `foogrep`, so use this with some caution).

most important processes: top

Dashboard that shows which processes are doing what. Use `q` to quit.

```
1 top
```

stop a process: kill

Send a signal to a process. Usually this is used to terminate misbehaving processes that won't stop of their own accord.

```
1 sleep 60 &
2 # The following invocation uses backticks to kill the process we just
3 # started. Copy/paste it into a separate text file if you can't distinguish the
4 # backticks from the standard quote.
5 kill `ps xw | grep sleep | cut -f1 -d ' ' | head -1`
```

Superuser

become root temporarily: sudo

Act as the root user for just one or a few commands.

```
1 sudo apt-get install -y git-core
```

become root: su

Actually become the root user. Sometimes you do this when it's too much of a pain to type `sudo` a lot, but you usually don't want to do this.

```
1 touch normal
2 sudo su
3 touch rootfile
4 ls -alrth normal rootfile # notice owner of rootfile
```

Storage and Finding

create an archive: tar

Make an archive of files.

```
1 mkdir genome
2 mv *ptt* genome/
3 tar -cvf genome.tar genome
```

compress files: gzip

Compress files. Can also view compressed `gzip` files without fully uncompressing them with `zcat`.

```
1 gzip genome.tar
2 md temp
3 cp genome.tar.gz temp
4 cd temp
5 tar -xzvf genome.tar.gz
6 cd ..
```

```
7 gunzip genome.tar.gz
8 rm -rf genome.tar genome temp
9 wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.fa.gz
10 zcat hs_ref_GRCh37.p10_chr22.fa.gz | nl | head -1000 | tail -20
```

find a file (non-indexed): find

Non-indexed search. Can be useful for iterating over all files in a subdirectory.

```
1 find /etc | nl
```

find a file (indexed): locate

For locate to work, updatedb must be operational. This runs by default on Ubuntu but you need to manually configure it for a mac.

```
1 sudo apt-get install -y locate
2 sudo updatedb # takes a little bit of time
3 locate fstab
```

system disk space: df

Very useful when working with large data sets.

```
1 df -Th
```

directory utilization: du

Use to determine which subdirectories are taking up a lot of disk space.

```
1 cd ~ubuntu
2 du --max-depth=1 -b | sort -k1 -rn
```

Intermediate Text Processing

searching within files: grep

Powerful command which is worth learning in great detail. Go through `grep --help` and try out many more of the options.

```
1 wget ftp://ftp.ncbi.nih.gov/genomes/Bacteria/\
2 Escherichia_coli_K_12_substr_W3110_uid161931/NC_007779.ptt
3 grep protein *ptt | wc -l # lines containing protein
4 grep -l Metazoa *ptt *gbk # print filenames which contain 'Metazoa'
5 grep -B 5 -A 5 Metazoa *gbk
6 grep 'JOURNAL.*' *gbk | sort | uniq
```

simple substitution: sed

Quick find/replace within a file. You should review this outstanding set of [useful sed one-liners](#). Here is a simple example of using `sed` to replace all instances of `kinase` with `STANFORD` in the first 10 lines of a `ptt` file, printing the results to STDOUT:

```
1 head *ptt | sed 's/kinase/STANFORD/g'
```

`sed` is also useful for quick cleanups of files. Suppose you have a file which was saved on Windows:

```
1 # Convert windows newlines into unix; use if you have such a file
2 wget http://startup-class.s3.amazonaws.com/windows-newline-file.csv
```

Viewing this file in `less` shows us some `^M` characters, which are Windows-format newlines (`\r`), different from Unix newlines (`\n`).

```
1 $ less windows-newline-file.csv
2 30,johnny@gmail.com,Johnny Walker,,^M1456,jim@gmail.com,Jim Beam,,^M2076,...
3 windows-newline-file.csv (END)
```

The `\r` is interpreted in Unix as a “carriage return”, so you can’t just `cat` these files. If you do, then the cursor moves back to the beginning of the line everytime it hits a `\r`, which means you don’t see much. Try it out:

```
1 cat windows-newline-file.csv
```

You won’t see anything except perhaps a blur. To understand what is going on in more detail, read [this post](#) and [this one](#). We can fix this issue with `sed` as follows:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sed 's/\r/\n/g' windows-newline-file.csv
2 30,johnny@gmail.com,Johnny Walker,,,
3 1456,jim@gmail.com,Jim Beam,,,
4 2076,jack@stanford.edu,Jack Daniels,,
```

Here `s/\r/\n/g` means “replace the `\r` with `\n` globally” in the file. Without the trailing `g` the `sed` command would just replace the first match. Note that by default `sed` just writes its output to STDOUT and does not modify the underlying file; if we actually want to modify the file in place, we would do this instead:

```
1 sed -i 's/\r/\n/g' windows-newline-file.csv
```

Note the `-i` flag for in-place replacement. Again, it is worth reviewing this list of [useful sed one-liners](#).

advanced substitution/short scripts: awk

Useful scripting language for working with tab-delimited text. Very fast for such purposes, intermediate size tool.

```
1 tail -n+4 *ptt | awk -F"\t" '{print $2, $3, $3 + 5}' | head
```

To get a feel for `awk`, review this list of [useful awk one liners](#). It is often the fastest way to operate on tab-delimited data before importation into a database.

Intermediate bash

Keyboard shortcuts

Given how much time you will spend using `bash`, it's important to learn all the [keyboard shortcuts](#).

- *Ctrl + A*: Go to the beginning of the line you are currently typing on
- *Ctrl + E*: Go to the end of the line you are currently typing on
- *Ctrl + F*: Forward one character.
- *Ctrl + B*: Backward one character.
- *Meta + F*: Move cursor forward one word on the current line
- *Meta + B*: Move cursor backward one word on the current line
- *Ctrl + P*: Previous command entered in history
- *Ctrl + N*: Next command entered in history
- *Ctrl + L*: Clears the screen, similar to the `clear` command
- *Ctrl + U*: Clears the line before the cursor position. If you are at the end of the line, clears the entire line.
- *Ctrl + H*: Same as backspace
- *Ctrl + R*: Lets you search through previously used commands
- *Ctrl + C*: Kill whatever you are running
- *Ctrl + D*: Exit the current shell
- *Ctrl + Z*: Puts whatever you are running into a suspended background process. `fg` restores it.
- *Ctrl + W*: Delete the word before the cursor
- *Ctrl + K*: Kill the line after the cursor
- *Ctrl + Y*: Yank from the kill ring

- *Ctrl + _*: Undo the last bash action (e.g. a yank or kill)
- *Ctrl + T*: Swap the last two characters before the cursor
- *Meta + T*: Swap the last two words before the cursor
- *Tab*: Auto-complete files and folder names

Let's go through a brief screencast of the most important shortcuts.

Backticks

Sometimes you want to use the results of a bash command as input to another command, but not via STDIN. In this case you can use [backticks](#):

```
1 echo `hostname`
2 echo "The date is "`date`
```

This is a useful technique to compose command line invocations when the results of one is the argument for another. For example, one command might return a hostname like `ec2-50-17-88-215.compute-1.amazonaws.com` which can then be passed in as the `-h` argument for another command via backticks.

Running processes in the background: `foo &`

Sometimes we have a long running process, and we want to execute other commands while it completes. We can put processes into the background with the ampersand symbol as follows.

```
1 sleep 50 &
2 # do other things
```

Here is how that actually looks at the command line:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 60 &
2 [1] 5472
3 ubuntu@domU-12-31-39-16-1C-96:~$ head -2 *ptt
4 Escherichia coli str. K-12 substr. W3110, complete genome - 1..4646332
5 4217 proteins
6 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep 5472
7 5472 pts/0    S      0:00 sleep 60
8 5475 pts/0    S+     0:00 grep --color=auto 5472
9 ubuntu@domU-12-31-39-16-1C-96:~$ 
10 ubuntu@domU-12-31-39-16-1C-96:~$ 
11 [1]+  Done                  sleep 60
```

Note that we see `[1] 5472` appear. That means there is one background process currently running (`[1]`) and that the ID of the background process we just spawned is `5472`. Note also at the end that when the process is done, this line appears:

```
1 [1]+ Done sleep 60
```

That indicates which process is done. For completeness, here is what it would look like if you had multiple background processes running simultaneously, and then waited for them all to end.

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 10 &
2 [1] 5479
3 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
4 [2] 5480
5 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 30 &
6 [3] 5481
7 ubuntu@domU-12-31-39-16-1C-96:~$ 
8 [1] Done sleep 10
9 [2]- Done sleep 20
10 [3]+ Done sleep 30
```

Execute commands from STDIN: xargs

This is a very powerful command but a bit confusing. It allows you to programmatically build up command lines, and can be useful for spawning parallel processes. It is commonly used in combination with the `find` or `ls` commands. Here is an example which lists all files under `/etc` ending in `.sh`, and then invokes `head -2` on each of them.

```
1 find /etc -name '*\.sh' | xargs head -2
```

Here is another example which invokes the `file` command on everything in `/etc/profile.d`, to print the file type:

```
1 ls /etc/profile.d | xargs file
```

A savvy reader will ask why one couldn't just do this:

```
1 file /etc/profile.d/*
```

Indeed, that will produce the same results in this case, and is feasible because only one directory is being listed. However, if there are a huge number of files in the directory, then `bash` will be unable to expand the `*`. This is common in many large-scale applications in search, social, genomics, etc and will give the dreaded `Argument list too long` error. In this case `xargs` is the perfect tool; see [here](#) for more.

Pipe and redirect: tee

Enables you to save intermediate stages in a pipeline. Here's an example:

```
1 ls | tee list.txt
```

The reason it is called `tee` is that it is like a “T-junction”, where it passes the data through while also serializing it to a file. It can be useful for backing up a file while simultaneously modifying it; see some [examples](#).

time any command: time

This is a `bash` built in useful for benchmarking commands.

```
1 time sleep 5
```

That should echo the fact that the command took (almost) exactly 5 seconds to complete. It is also useful to know about a more sophisticated GNU `time` command, which can output a ton of useful information, including maximum memory consumption; extremely useful to know before putting a job on a large compute cluster. You can invoke it with `/usr/bin/time`; see documentation for Ubuntu 12 LTS [here](#).

Summary

We covered quite a few command line programs here, but nothing substitutes for running them on your own and looking at the online help and flags. For example, with `head` you can do `head --help`, `man head`, and `info head` to get more information.

Linux Development Environment

In this lecture we will set up a Linux environment for writing and debugging SSJS code using `screen`, `emacs`, and `git`. Then we will show how to set up a new machine in a scriptable and reproducible way via `setup.git` and `dotfiles.git`.

Development Environment

`screen`: Tab manager for remote sessions

When Google Chrome or Firefox crashes, it offers to restore your tabs upon reboot. The `screen` program is similar: when your network connection to the remote host drops out, you can reconnect and resume your session. This is more than a convenience; for any long-running computation, `screen` (or background processes) is the only way to do anything nontrivial. For the purposes of this class, let's use a custom `.screenrc` file which configures `screen` to play well with the `emacs` text editor (to be introduced shortly). You can install this config file as follows:

```
1 $ cd $HOME
2 $ wget spark-public.s3.amazonaws.com/startup/code/screenrc -O .screenrc
3 $ head .screenrc
4 $ screen
```

Let's go through an interactive session to see the basics of `screen`. If you want to learn even more about `screen`, go through this excellent [tutorial](#) and [video tutorial](#); if you want to use `tmux` instead, [this](#) is a reasonable starting place.

`emacs`: An environment for writing code

What is a text editor? A text editor like `emacs` or `nano` is different from a word processor like Microsoft Word in that it allows you to directly manipulate the raw, unadorned string of bytes that make up a file. That's why text editors are used by programmers, while word processors are employed by end users (Figure 1).

Many people use GUI-based text editors like [Textmate](#), [Sublime Text](#), [Visual Studio](#), [XCode](#), or [Eclipse](#). These editors have several good properties, but don't support all of the following properties simultaneously:

- free
- open-source
- highly configurable

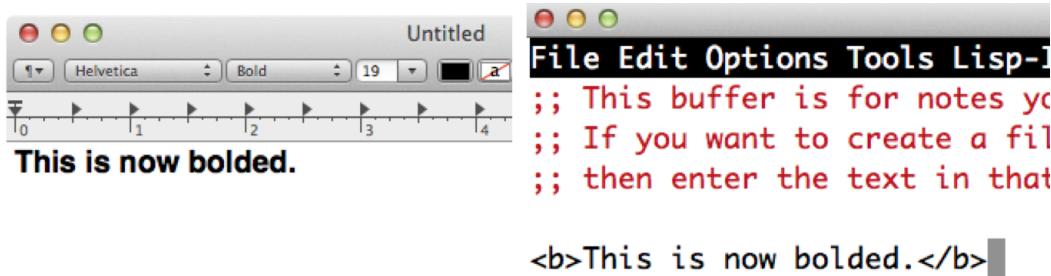


Figure 1: A word processor (left) vs. a text-editor (right). The character escapes to bold text are explicit in a text editor and hidden in a word processor.

- large dev community
- editing modes for virtually every language
- external tool integration
- ultra cross platform
- embeddable in other programs
- fast boot time
- lightweight memory footprint

Two editors that do have all these features are `vim` (vim.org) and `emacs` (gnu.org/software/emacs). If you are doing a technology startup, you probably¹ want to learn one of these two. In general, `vim` is more suitable for sysadmins or people who must manually edit many little files. It has excellent defaults and is very fast for pure text-editing. By contrast, `emacs` is greatly preferred for the data scientist or academic due to features like `orgmode` and strong read-evaluate-print-loop (`REPL`) integration for interactive debugging sessions. For intermediate tasks like web development, one can use either editor, but the `REPL` integration features of `emacs` carry the day in our view.²

OS X and Windows: Control and Meta Keys. Many keyboard sequences³ in Unix depend on pressing multiple keys simultaneously. We use the following notation for key combinations using Control or Meta:

¹Of course there are smart people who don't use `emacs` or `vim`; Jeff Atwood of StackOverflow used [Visual Studio](#) and built it on the .NET stack, and DHH of Ruby on Rails fame famously uses [TextMate](#). Still, even Jeff Atwood used Ruby for his [latest project](#), and the top technology companies ([Google](#), [Facebook](#), et alia) tend to use `vim` or `emacs` *über alles*.

²One editor out there that might be a serious contender to the `vim` and `emacs` duopoly is the new [Light Table](#) editor by Chris Granger. A key question will be whether it can amass the kind of open-source community that `emacs` and `vim` have built, or alternatively whether its business model will provide enough funds to do major development in-house.

³The `emacs` keybindings are used by default in any application that uses the `readline` library (like `bash`), while `vim` keybindings are used in commands like `less`, `top`, and `visudo`. The major advantage of investing in one of these editors is that you can get exactly the same editing environment on your local machine or any server that you connect to regularly.

- C-a: press Control and A simultaneously
- M-d: press Meta and D simultaneously
- C-u 7: press Control and U together, then press 7
- C-c C-e: press Control and C simultaneously, let go, and then press Control and E simultaneously

Given how frequent this is, to use `emacs`, `bash`, and `screen` with maximum efficiency you will want to customize your control and meta keys to make them more accessible.

1. *Windows: Keyboard Remapper:* On Windows you will want to use a [keyboard remapper](#) and possibly use [Putty](#) if you have troubles with Cygwin.
2. *Mac: Terminal and Keyboard Preferences:* On a Mac, you can configure Terminal Preferences to have “option as meta” and Keyboard Preferences to swap the [option](#) and [command](#) keys, and set Caps Lock to Control, as shown in Figures 2 and 3.

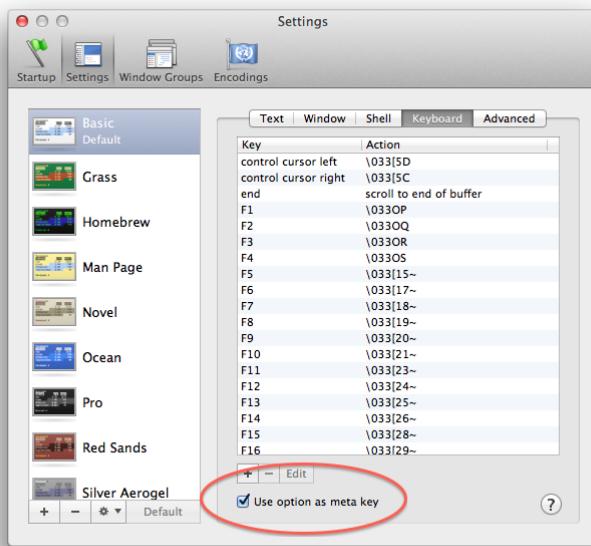


Figure 2: Click *Settings* (in the top row of buttons) and then “Keyboard”. Click “Use option as meta key”. You’ll need this for `emacs`.

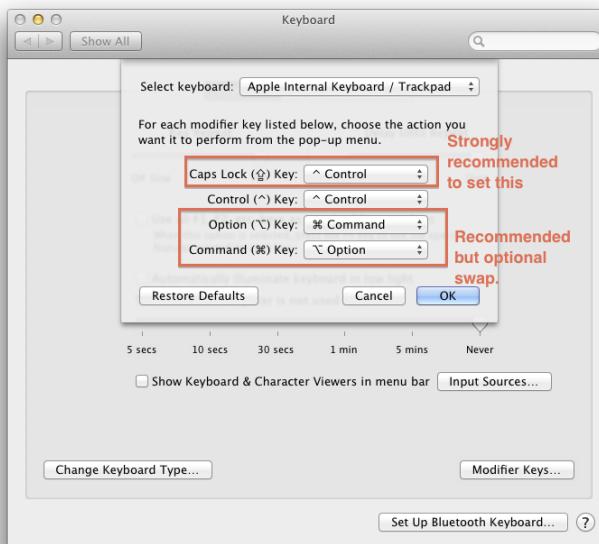


Figure 3: Switch Caps-lock and Control, and (at your discretion) switch Option and Command. The latter will make emacs much easier to use as you'll be hitting Option/Meta a lot.

When complete, now we are done with the preliminaries of configuring our local keyboard mappings.

Installing Emacs. Let's now install the latest version of `emacs` using these instructions⁴ for [Ubuntu](#):

```
1 sudo apt-add-repository -y ppa:cassou/emacs
2 sudo apt-get update
3 sudo apt-get install -y emacs24 emacs24-el emacs24-common-non-dfsg
```

Emacs basics. To get started with `emacs`, watch the video lecture for this section for an interactive screencast. This shows you how to launch and quit, and how to start the online tutorial. Notice in particular the command combinations streaming by in the lower right corner. The first thing you want to do is go through `M-x help-with-tutorial` and start learning the keyboard shortcuts. With `emacs`, you will never again need to remove your hands from the touch typing position. You will also want to keep this [reference card](#) handy as you are learning the commands.

Emacs and the node.JS REPL. Now we get to the real payoff of doing all this setup. The real win of `emacs` comes from the ability to run an interactive REPL next to your code on any machine. First, execute the following commands to download some emacs-specific configuration (see Scripting the setup of a developer environment).

```
1 cd $HOME
2 git clone https://github.com/startup-class/dotfiles.git
3 ln -sb dotfiles/.screenrc .
4 ln -sb dotfiles/.bash_profile .
5 ln -sb dotfiles/.bashrc .
6 ln -sb dotfiles/.bashrc_custom .
7 mv .emacs.d .emacs.d~
8 ln -s dotfiles/.emacs.d .
```

The configuration files we just symlinked into `~/.emacs.d` enable us to write and debug JS code interactively. Let's try this out with the Fibonacci code from last time:

```
1 wget https://spark-public.s3.amazonaws.com/startup/code/fibonacci.js
2 emacs -nw fibonacci.js
```

When in `emacs`, hit `C-c!` to bring up a prompt in the `*js` buffer. Then hit `C-cC-j` to send the current line from the `hello.js` buffer to the prompt in the `*js*` buffer, or select a set of lines and hit `C-cC-r` to send an entire region. In this manner you can write code and then

⁴If you want to run `emacs` locally, on OS X you can install this from [emacsformacosx.com](#). On Windows you can install it from [here](#) or from within the Cygwin [package installer](#); see also [here](#) and [here](#) for more tips.

rapidly debug it⁵ with just a few keystrokes. Watch the video lecture for this section for details.

Emacs and jshint. Javascript is a language that has many idiosyncracies and stylistic warts; Douglas Crockford's book [Javascript: The Good Parts](#) and [JS Wat](#) go through many of these issues. Perhaps the easiest way to confirm that you haven't done something silly (like missing a semicolon) is to integrate [jshint](#) ([jshint.com](#)) into `emacs` as a compilation tool. First let's install `jshint` globally:

```
1 npm install -g jshint
```

Using the configuration from the last time, we can now hit `C-cC-u` to automatically check our JS code for errors. Watch the screencast in this section of the lecture to see how to interactively find and debug JS issues within `emacs` with `jshint`.

Troubleshooting Emacs. Emacs is controlled by Emacs Lisp ([1](#), [2](#)), which is a language in its own right. If you have problems loading up `emacs` at any time, execute the command `emacs --debug-init`. That will tell you the offending line in your `emacs` configuration. You can also use `M-x<RET>ielm` to get an interactive Elisp REPl prompt where you can type in commands, `M-x<RET>describe-mode` to see the commands in the current mode, and `M-x<RET>apropos` to search for help on a given topic.

Going further with Emacs. We have only scratched the surface. There are many things you can do with `emacs`, including:

1. *Pushing and pulling with magit*: You can always push and pull to `git` from the command line. But it's a little easier if you use `magit` to do so. See [here](#).
2. *Searching into a function with etags*: Once you get to the point that you have multifile codebases, you will want to jump to the point of a given function definition. You can do this by generating an index file with the `etags` command; see [here](#).
3. *Tab autocomplete and snippets*: With `hippie-expand` and `yasnippet`, you can use tab to autocomplete any symbol which is present in any open buffer. This is fantastic for working with long variable or function names, and reduces the chances of accidentally misspelling the name of a constant.

And these are only for coding; Emacs also has tools like:

- `org-mode` for note-taking and reproducible research.
- `gnus` for rapidly reading email.
- `dired-mode` for directories
- `tramp` for remote access

⁵This can be generalized to any language; indeed, we have code for setting up a similar REPL environment for many other languages and interpreters (R, Matlab, python/ipython, haskell/ghc), using the `emacs comint-mode` features ([1](#), [2](#), [3](#), [4](#)).

- patch-mode for looking at diffs

Indeed, virtually every programming task has an Emacs major mode. And if you master Elisp (1, 2) you can configure your development environment such that you can do anything in one or two keystrokes from within `emacs`.

Emacs Summary. To summarize, we just explored the basics of how to launch `emacs` at the command line, opening and switching windows, finding online help (with M-x `describe-mode`, M-x `describe-command`, M-x `apropos`), opening/creating/saving files, and the use of `emacs` for editing SSJS with a REPL. You should practice editing files and print the `emacs` reference card to make sure you master the basics.

git: Distributed Version Control Systems (DVCS)

To understand the purpose of `git` and Github, think about your very first program. In the process of writing it, you likely saved multiple revisions like `foo.py`, `foo_v2.py`, and `foo_v9_final.py`. Not only does this take up a great deal of redundant space, using separate files of this kind makes it difficult to find where you kept a particular function or key algorithm. The solution is something called *version control*. Essentially, all previous versions of your code are stored, and you can bring up any past version by running commands on `foo.py` rather than cluttering your directory with past versions.

Version control becomes even more important when you work in a multiplayer environment, where several people are editing the same `foo.py` file in mutually incompatible ways at the same time on tight deadlines. In 2005, Linus Torvalds (the creator of Linux) developed a program called `git` that elegantly solved many of these problems. `git` is a so-called *distributed version control system*, a tool for coordinating many simultaneous edits on the same project where byte-level resolution matters (e.g. the difference between `x=0` and `x=1` is often profound in a program). `git` replaces and supersedes previous version control programs like `cvs` and `svn`, which lack distributed features (briefly, tools for highly multiplayer programming), and is considerably faster than comparable DVCS like `hg`.

The SHA-1 hash. To understand `git`, one needs to understand a bit about the [SHA-1](#) hash. Briefly, any file in a computer can be thought of as a series of bytes, each of which is 8 bits. If you put these bytes from left to right, this means all computer files can be thought of as very large numbers represented in binary (base-2) format. Cryptographers have come up with a very interesting function called SHA-1 which has the following curious property: any binary number, up to 2^{64} bits, can be rapidly mapped to a 160 bit (20 byte) number that we can visualize as a 40 character long number in hexadecimal (base-16) format. Here is an example using node's `crypto` module:

```

1 $ node
2 > var crypto = require('crypto')
3 > crypto.createHash('sha1').update("Hello world.", 'ascii').digest('hex')
4 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
```

Moreover, even binary numbers which are very close map to completely different 20 byte SHA-1 values, which means $\text{SHA-1}(x)$ is very different from most “normal” functions like

$\cos(x)$ or e^x for which you can expect [continuity](#). The interesting thing is that despite the fact that there are a finite number of SHA-1 values (specifically, 2^{160} possible values) one can assume that if two numbers are unequal, then their hashes are also extremely likely to be unequal. Roughly speaking this means:

$$x_1 \neq x_2 \rightarrow P(\text{SHA-1}(x_1) \neq \text{SHA-1}(x_2)) \geq .9999\dots$$

Here is how that works out in practice. Note that even small perturbations to the “Hello world.” string, such as adding spaces or deleting periods, result in completely different SHA-1 values.

```

1 $ node
2 > var crypto = require('crypto')
3 > var sha1 = function(str) {
4   return crypto.createHash('sha1').update(str, 'ascii').digest('hex');
5 }
6 > sha1("Hello world.")
7 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
8 > sha1("Hello world. ")
9 '43a898d120ad380a058c631e9518c1f57d9c4edb'
10 > sha1("Hello world")
11 '7b502c3a1f48c8609ae212cdfb639dee39673f5e'
12 > sha1(" Hello world.")
13 '258266d16638ad59ac0efab8373a03b69b76e821'
```

Because a hash can be assumed to map 1-to-1 to a file, rather than carting around the full file’s contents to distinguish it, you can just use the hash. Indeed, you can just use a hash to uniquely⁶ identify *any* string of bits less than 2^{64} bits in length. Keep that concept in mind; you will need it in a second.

A first git session. Now let’s go through a relatively simple interactive session to create a small `git` repository (“repo”). Type in these commands at your Terminal; they should also work on a local Mac, except for the `apt-get` invocations.

```

1 sudo apt-get install -y git-core
2 mkdir myrepo
3 cd myrepo
4 git config --global user.name "John Smith"
5 git config --global user.email "example@stanford.edu"
6 git init
7 # Initialized empty Git repository in /home/ubuntu/myrepo/.git/
8 git status
9 echo -e 'line1\nline2' > file.txt
10 git status
11 git add file.txt
```

⁶Is it really unique? No. But finding a collision in SHA-1 is hard, and you can show that it’s much more likely that other parts of your program will fail than that a nontrivial collision in SHA-1 will occur.

```

12 git status
13 git commit -m "Added first file"
14 git log
15 echo -e "line3" >> file.txt
16 git status
17 git diff file.txt
18 git add file.txt
19 git commit -m "Added a new line to the file."
20 git log
21 git log -p
22 git log -p --color

```

That is pretty cool. We just created a repository, made a dummy file, and added it to the repository. Then we modified it and looked at the repo again, and we could see exactly which lines changed in the file. Note in particular what we saw when we did `git log`. See those alphanumeric strings after `commit`? Yes, those are SHA-1 hashes of the entire commit, conceptually represented as a single string of bytes, a “diff” that represents the update since the last commit.

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git log
2 commit 5c8c9efc99ad084af617ca38446a1d69ae38635d
3 Author: Balaji S <balajis@stanford.edu>
4 Date: Thu Jan 17 16:32:37 2013 +0000
5
6     Added a new line to the file.
7
8 commit 1b6475505b1435258f6474245b5161a7684b7e2e
9 Author: Balaji S <balajis@stanford.edu>
10 Date: Thu Jan 17 16:31:23 2013 +0000
11
12     Added first file.

```

What is very interesting is that `git` uses SHA-1 all the way down. We can copy and paste the SHA-1 identifier of a commit and pass it as an argument to a `git` command (this is a very frequent operation). Notice that `file.txt` itself has its own SHA-1.

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git ls-tree 5c8c9efc99ad084af617ca38446a1d69ae38635d
2 100644 blob 83db48f84ec878fbfb30b46d16630e944e34f205 file.txt

```

And we can use this to pull out that particular file from within `git`'s bowels:

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git cat-file -p 83db48f84ec878fbfb30b46d16630e944e34f205
2 line1
3 line2
4 line3

```

The point here is that git tracks *everything* with a SHA-1 identifier. Every alteration you make to the source code tree is tracked and can be uniquely addressed and manipulated with a SHA-1 identifier. This allows very interesting kinds of computations using `git`.

Pushing to Github.com While you can use `git` purely in this kind of local mode, to collaborate with others and protect against data loss you'll want to set up a centralized repository. You can certainly host your own repository, but it will be bare bones and yet another service to maintain. Or you can use one of many `git hosting services`, such as Bitbucket, Google Code, Sourceforge, or Github. The last will be our choice for this class. You can think of github.com as a web frontend to `git`, but it is now much more than that. It's one of the largest open source communities on the web, and the github.com website is a big deal⁷ in its own right. With that background, let's create a web version of our repo and push this content to github. Go to github.com/new and initialize a new repo as shown:

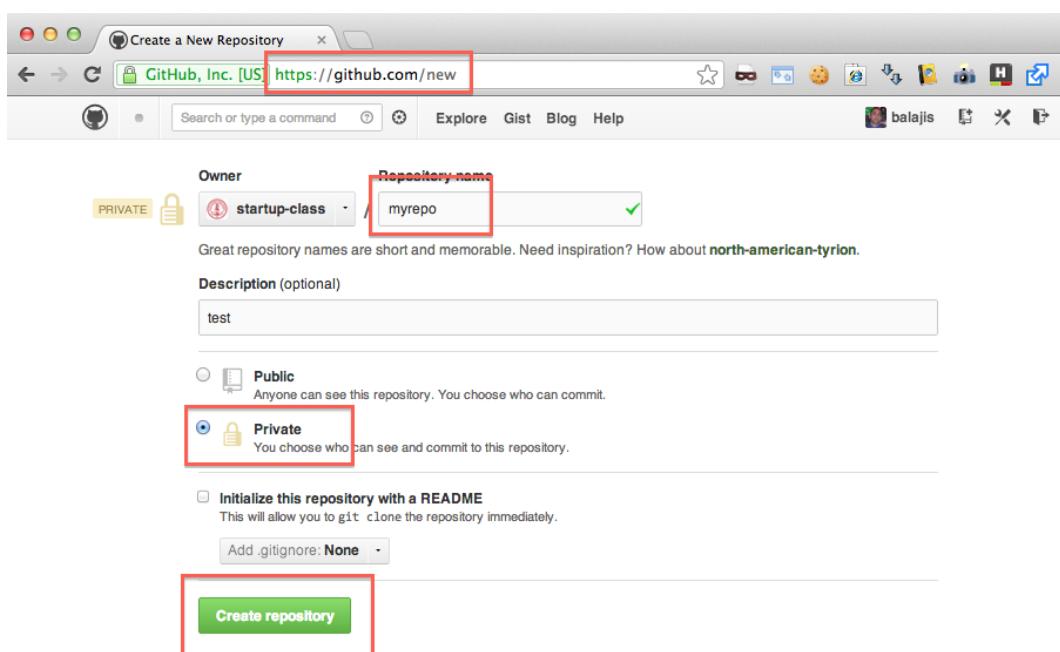


Figure 4: Creating a new repo at github.com/new.

⁷At some point you should go through github's [online tutorial](#) content to get even more comfortable with git; the learn.github.com/p/intro.html page is a good place to begin, following along with the commands offline. This will complement the instructions in this section. Once you've done that, go to help.github.com and do the four sections at the top to get familiar with how the github.com site adds to the `git` experience.

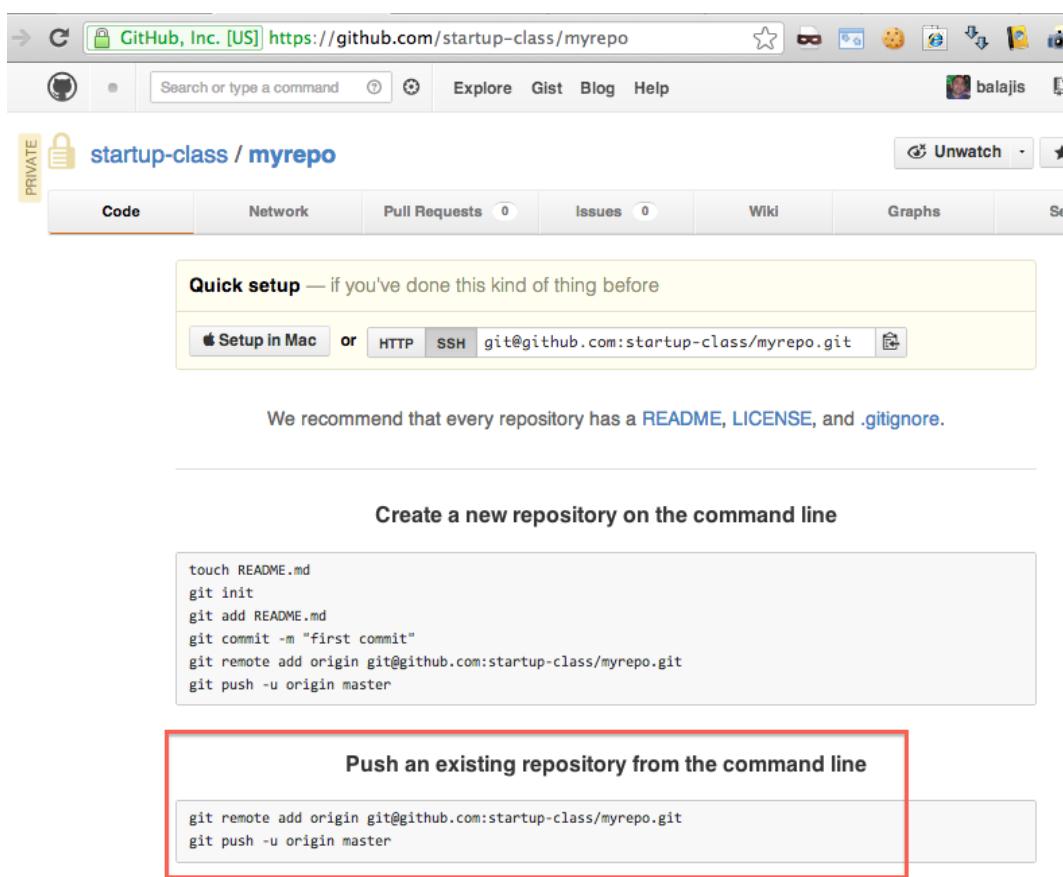


Figure 5: Instructions after a new repo is created. Copy the first line in the highlighted box.

Now come back to EC2 and run these commands:

```
1 git remote add origin git@github.com:startup-class/myrepo.git  
2 git push -u origin master # won't work
```

The second command won't work. We need to generate some `ssh` keys so that github knows that we have access rights as [documented here](#). First, run the following command:

```
1 cd $HOME  
2 ssh-keygen -t rsa -C "foo@stanford.edu"
```

Here's what that looks like:

```
ubuntu@ip-10-196-107-40:~/myrepo$ cd  
ubuntu@ip-10-196-107-40:~$ ssh-keygen -t rsa -C "balajis@stanford.edu"  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.  
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.  
The key fingerprint is:  
72:6a:f4 balajis@stanford.edu  
The key's randomart image is:  
+--[ RSA 2048]----+  
| ..o |  
| . o |  
| . |  
| . |  
| c |  
| . |  
| . |  
| . . |  
+-----+  
ubuntu@ip-10-196-107-40:~$
```

Hit enter twice when prompted for a passphrase.

Figure 6: Generating an `ssh-key` pair.

Next, use `cat` and copy that text exactly as shown:

```
ubuntu@ip-10-196-107-40:~$ cd .ssh/
ubuntu@ip-10-196-107-40:~/ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDNNgDElEra5AZAVrkcj6gYYAnZkEefuMn/V0py6yfL4sdU8+
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
[REDACTED]
Evi5005u2+qy1D5fusATx27y121nsu1yM balajis@stanford.edu
ubuntu@ip-10-196-107-40:~/ssh$
```

Figure 7: Copy the text exactly from the first “s” in “ssh-rsa” to the last “u” in “stanford.edu”.

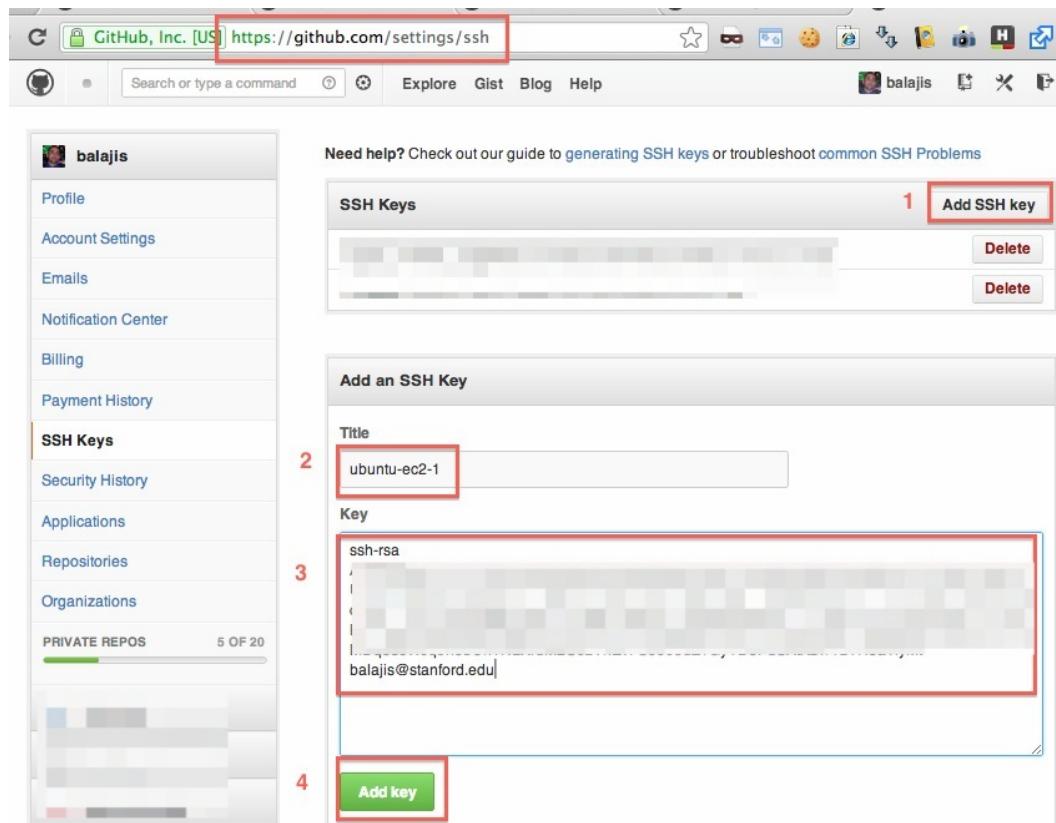


Figure 8: Now go to `github.com/settings/ssh`, click “Add SSH Key” in the top right, enter in a Title and then paste into the Key field. Hit “Add Key”.

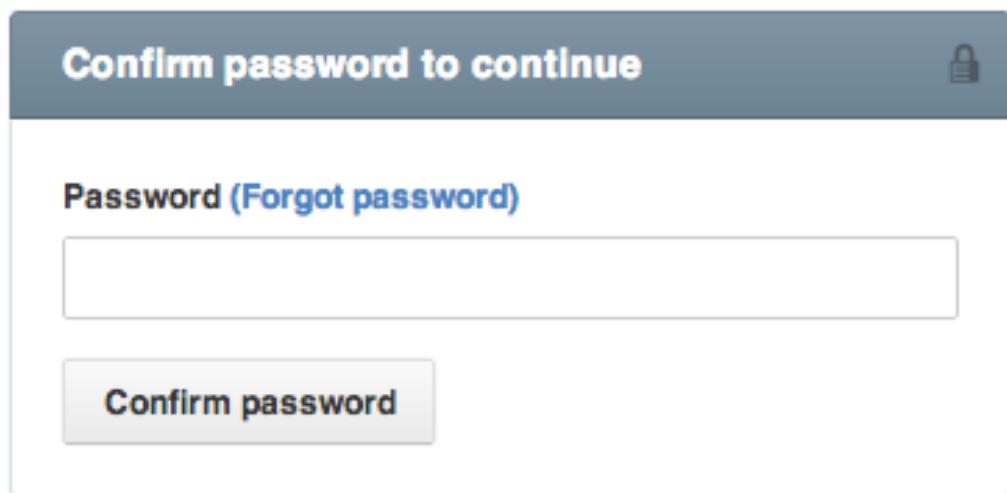


Figure 9: You will see a prompt for your github password.

Now come back to the command line and type in:

```
1 ssh -T git@github.com
```

```
|ubuntu@ip-10-196-107-40:~/ssh$ ssh -T git@github.com  
Hi balajis! You've successfully authenticated, but GitHub does not provide shell access.
```

Figure 10: You should see this if you've added the SSH key successfully.

Now, finally you can do this:

```
1 git push -u origin master # will work after Add Key
```

```
ubuntu@ip-10-196-107-40:~/ssh$ cd ..  
ubuntu@ip-10-196-107-40:~$ cd myrepo/  
ubuntu@ip-10-196-107-40:~/myrepo$ git push -u origin master  
Counting objects: 6, done.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (6/6), 458 bytes, done.  
Total 6 (delta 0), reused 0 (delta 0)  
To git@github.com:startup-class/myrepo.git  
 * [new branch]      master -> master  
Branch master set up to track remote branch master from origin.
```

Figure 11: Now you can run the command to push to github.

And if you go to the URL you just set up, at `https://github.com/$USERNAME/myrepo`, you will see a git repository. If you go to `https://github.com/$USERNAME/myrepo/commits/master` in particular, you will see your Gravatar next to your commits. Awesome. So we've just linked together EC2, Linux, git, Github, and Gravatar.

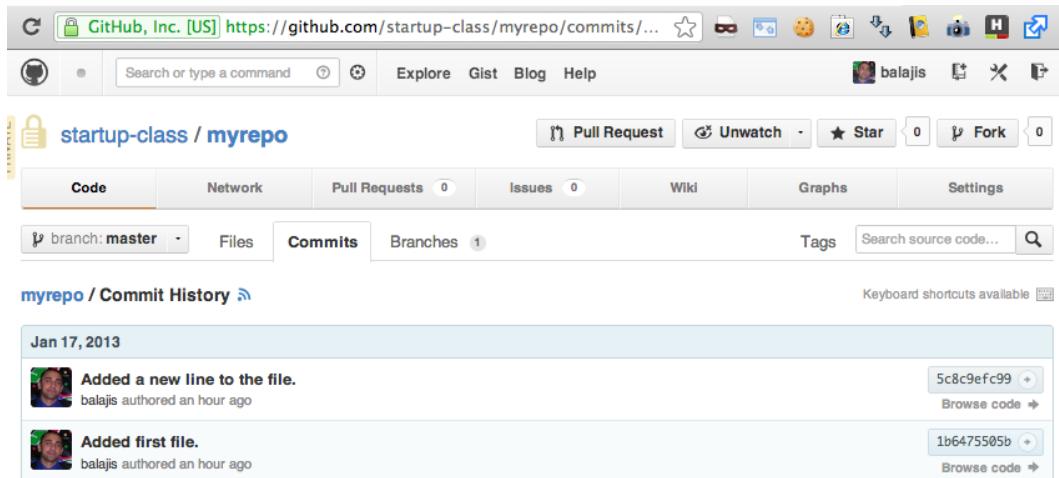


Figure 12: Now you can see your command line commits on the web, and share them with classmates.

Congratulations. That's the basics of git right there: creating a repository, adding files, making commits, and pushing to a remote repository. We will be using git quite a bit over the course, and will get into intermediate usage soon, but in the interim it will be worth your while to go a bit deeper on your own time. There are a number of excellent references, including the official [book](#) and [videos](#) and this [reference](#) for advanced users.

Managing Setup and Configuration as Code

The concept of `setup.git` and `dotfiles.git`

One of the things you may have started to realize is that it is not completely trivial to set up a working environment in which to write, edit, and debug code. We keep running `apt-get` commands all the time, and we needed to download some files to get `emacs` to work. A key concept is that the shell commands issued during the setup process constitute code in their own right, as do the configuration files used for `bash`, `screen`, `emacs`, and the like. By saving these files themselves in a `git` repository we can rebuild a development environment with one command. Indeed, this idea of scriptable infrastructure is one of the core ideas behind *DevOps* (1, 2), and there are tools like `chef` and `puppet` that facilitate automation of very sophisticated deployments. But we'll start simple with just a shell script.

Scripting the setup of an EC2 instance

Let's try this out with `setup.git` on a fresh EC2 instance.

```
1 cd $HOME
2 sudo apt-get install -y git-core
3 git clone https://github.com/startup-class/setup.git
4 ./setup/setup.sh
```

And just like that, we just ran a script that installed everything we've been using on the local machine.

Scripting the setup of a developer environment

Within `setup.git` are the commands to pull down your developer environment via a second repo called `dotfiles.git`. If you look at `setup.sh`, it calls the commands we ran previously:

```
1 cd $HOME
2 git clone https://github.com/startup-class/dotfiles.git
3 ln -sb dotfiles/.screenrc .
4 ln -sb dotfiles/.bash_profile .
5 ln -sb dotfiles/.bashrc .
6 ln -sb dotfiles/.bashrc_custom .
7 mv .emacs.d .emacs.d~
8 ln -s dotfiles/.emacs.d .
```

To understand what is happening here: the `ln -s` command is used to symbolically link in configuration files to the home directory, and the `-b` flag indicates that we will make a backup of any existing files. Thus `.bashrc\~` is created in the directory if a `.bashrc` already exists.. However, the `-b` flag only works on files, not directories, so we need to move `.emacs.d` to `.emacs.d~` with an explicit `mv` command. After executing these commands, if you now log out (by hitting `C-d` or typing `exit` in any directory) and log back in to the instance, you will see your prompt transform into something more useful:

```
ubuntu@ip-10-204-245-82:~$ git clone git://github.com/startup-class/dotfiles.git
Cloning into 'dotfiles'...
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 18 (delta 3), reused 18 (delta 3)
Receiving objects: 100% (18/18), 9.85 KiB, done.
Resolving deltas: 100% (3/3), done.
ubuntu@ip-10-204-245-82:~$ cd dotfiles/
```

Figure 13: Prompt before `.bashrc` and `.bash_profile` configuration

The screenshot shows a terminal window titled "ubuntu@ip-10-204-245-82:~/dotfiles — ssh — 114x30 — %1". The terminal content is as follows:

```
[ubuntu@ip-10-204-245-82:~]$ ls
dotfiles
[ubuntu@ip-10-204-245-82:~]$ ll
total 40
-rw-r--r-- 1 ubuntu ubuntu 675 Apr  3 2012 .profile
-rw-r--r-- 1 ubuntu ubuntu 3486 Apr  3 2012 .bashrc~
-rw-r--r-- 1 ubuntu ubuntu 220 Apr  3 2012 .bash_logout
drwxr-xr-x 3 root  root  4096 Jan 24 07:07 ../
drwx----- 2 ubuntu ubuntu 4096 Feb  7 14:35 .cache/
-rw-r--r-- 1 ubuntu ubuntu    0 Feb  7 15:27 .sudo_as_admin_successful
drwx----- 2 ubuntu ubuntu 4096 Feb  7 15:27 .ssh/
drwxr-xr-x 3 ubuntu ubuntu 4096 Feb  7 15:28 dotfiles/
lrwxrwxrwx 1 ubuntu ubuntu   22 Feb  7 15:28 .bash_profile -> dotfiles/.bash_profile
lrwxrwxrwx 1 ubuntu ubuntu   16 Feb  7 15:28 .bashrc -> dotfiles/.bashrc
lrwxrwxrwx 1 ubuntu ubuntu   23 Feb  7 15:28 .bashrc_custom -> dotfiles/.bashrc_custom
-rw----- 1 ubuntu ubuntu  323 Feb  7 15:29 .bash_history
drwxr-xr-x 5 ubuntu ubuntu 4096 Feb  7 15:29 ./
-rw-rw-r-- 1 ubuntu ubuntu 1026 Feb  7 15:31 .bash_eternal_history
[ubuntu@ip-10-204-245-82:~]$ cd dotfiles/
[ubuntu@ip-10-204-245-82:~/dotfiles]$ ls
README.md
[ubuntu@ip-10-204-245-82:~/dotfiles]$
```

Figure 14: Prompt after `.bashrc` and `.bash_profile` configuration

Note that the username, hostname, and current directory are now displayed in the prompt. This provides context on your current environment. Go back and reread previous lectures if you need to refresh on bash. In addition to this, we have defined a `.bash_eternal_history` file in your home directory which will record all past commands, along with a few useful aliases:

```

ubuntu@ip-10-204-245-82:~$ head .bash_etal_history
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:11 2013 1360250951      /home/ubuntu  13  ln -sb
dotfiles/.bashrc_custom .
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:15 2013 1360250955      /home/ubuntu  14  alias
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:20 2013 1360250960      /home/ubuntu  15  em
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:27 2013 1360250967      /home/ubuntu  16  ll
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:35 2013 1360250975      /home/ubuntu  17  less .b
ash_etal_history
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:47 2013 1360250987      /home/ubuntu  18  less .b
ashrc
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:51 2013 1360250991      /home/ubuntu/dotfiles  19
cd dotfiles/
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:52 2013 1360250992      /home/ubuntu/dotfiles  20
ll
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:29:53 2013 1360250993      /home/ubuntu  21 ..
4252  ubuntu ip-10-204-245-82      screen  02/07/13 15:31:27 2013 1360251087      /home/ubuntu  22  ls
[ubuntu@ip-10-204-245-82:~]$
```

Figure 15: Bash eternal history

```

ubuntu@ip-10-204-245-82:~$ alias
alias ..='cd ..'
alias ...='cd ..;cd ..'
alias cl='clear'
alias cp='cp -i'
alias dir='ls --color=auto --format=vertical'
alias du='du -ch --max-depth=1'
alias em='emacs -nw'
alias eqq='emacs -nw -Q'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alrtF --color'
alias m='less'
alias md='mkdir'
alias mv='mv -i'
alias node='env NODE_NO_READLINE=1 rlwrap node'
alias rm='rm -i'
alias treeacl='tree -A -C -L 2'
alias vdir='ls --color=auto --format=long'
[ubuntu@ip-10-204-245-82:~]$
```

Figure 16: List of aliases

You can define more aliases in your `~/.bashrc_custom` file. Feel free to fork the `dotfiles.git` repository into your own github account and edit. The last thing to note is that we added these lines to your `~/.bashrc`:

```
1 # 2.6) node.js nvm
2 # http://nodejs.org/api/repl.html#repl_repl
3 # sudo apt-get install -y rlwrap
4 alias node="env NODE_NO_READLINE=1 rlwrap node"
5 alias node_repl='node -e "require('`repl`').start({ignoreUndefined: true})"'
6 export NODE_DISABLE_COLORS=1
7 if [ -s ~/.nvm/nvm.sh ]; then
8     NVM_DIR=~/.nvm
9     source ~/.nvm/nvm.sh
10    nvm use v0.10.12 &> /dev/null # quiet the nvm use command
11 fi
```

The `alias` line in this configuration snippet redefines the `node` command to use `rlwrap`, which makes it much easier to retrieve and edit previous commands in `node` ([see here for more](#)). It also defines a `node_repl` convenience command which turns off the default `node` practice of echoing `undefined` to the prompt for commands that don't return anything, which can be a bit distracting. The other lines turn off coloration in the `node` interpreter and automatically use `nvm` to initialize the latest version of `node`.

As an exercise for the reader, you can [fork](#) and update your `dotfiles.git` to further customize your dev environment. And you can even further automate things by using the [EC2 command line tools \(zip\)](#) from AWS, which in conjunction with a [user-data-script](#) would let you instantiate an EC2 instance from your local Terminal.app (OS X) or Cygwin (Windows) command line rather than going to [aws.amazon.com](#) and clicking “Launch Instance”. But at this point you get the idea: any setup or configuration of a machine is itself code and should be treated like code, by keeping the install scripts and config files under version control in a `git` repository.

Summary

In this lecture you set up a development environment on Ubuntu using `emacs` and `screen`, learned `git` and pushed some of your first repos to github, and learned how to orchestrate a reproducible dev machine setup.

Market Research, Wireframing, and Design

Idea, Execution, and Market

We begin with an interesting comment on [Quora](#), which we've slightly modified:

An idea is not a mockup
A mockup is not a prototype
A prototype is not a program
A program is not a product
A product is not a business
And a business is not profits

This doubles as a kind of state machine for your ideas as you push them through to production; at each stage many ideas fail to make it to the next one, both because of time requirements and because of some heretofore nonobvious [flaw](#) (Table 1). In particular, while you should be able to get to the *program* stage with purely technical training, the last three stages are in many ways the most challenging for pure scientists and engineers, because they generally aren't well taught in school. Let's see if we can explain why.

Table 1: *The idea state machine. Any successful entrepreneur has not one good idea, but a profusion of good ideas that lie abandoned at the beginning of the state machine, at the base of this pyramid. Moreover, many more napkin ideas (e.g. [micropayments](#)) have fatal flaws that only become apparent in the implementation stage. It is only through actually pushing ideas through the state machine that one realizes whether they are good or not. It's useful to actually keep a single Google Spreadsheet or orgmode document with all your ideas, organized by stage in this manner. Expect most to stay at the idea stage!*

Stage	What's required to complete?	Minimum Time
Idea	Napkin drawing of billion dollar concept	1 minute
Mockup	Wireframe with all user screens	1 day+
Prototype	Ugly hack that works for single major use case	1 weekend+
Program	Clean code that works for all use cases, with tests	2-4 weeks+
Product	Design, copywriting, pricing, physical components	3-6 months+
Business	Incorporation, regulatory filings, payroll, ...	6-12 months+
Profits	Sell product for more than it costs you to make it	1 year-∞

Idea

When it comes to starting a business, the conventional wisdom is that the idea is everything. This is why US politicians and patent lawyers recently pushed for the [America Invents Act](#),

why many people think they could have invented Facebook¹, and why reputed patent troll **Intellectual Ventures** is able to make money from **2000 shell companies** without ever shipping a product. In other words, the conventional wisdom is that with the right idea, it's *just* a matter of details to bring it to market and make a billion bucks. This is how the general public thinks technological innovation happens.

Execution

An alternate view comes from **Bob Metcalfe**, the inventor of Ethernet and founder of 3Com:

Metcalfe says his proudest accomplishment at the company was as head of sales and marketing. He claims credit for bringing revenue from zero to more than \$1 million a month by 1984. And he's careful to point out that it was this aptitude - not his skill as an inventor - that earned him his fortune.

"Flocks of MIT engineers come over here," Metcalfe tells me, leading me up the back staircase at Beacon Street. "I love them, so I invite them. They look at this and say, 'Wow! What a great house! I want to invent something like Ethernet.'" The walls of the narrow stairway are lined with photos and framed documents, like the first stock certificate issued at 3Com, four Ethernet patents, a photo of Metcalfe and Boggs, and articles Metcalfe has written for The New York Times and The Wall Street Journal.

"I have to sit 'em down for an hour and say, 'No, I don't have this house because I invented Ethernet. I have this house because I went to Cleveland and Schenectady and places like that. I sold Ethernet for a decade. That's why I have this house. It had nothing to do with that brainstorm in 1973.'"

In some ways this has become the new conventional wisdom: it's not the idea, it's the execution. You'll see variants of this phrase **repeated constantly** on Hacker News. The rationale here is that it's easy to come up with a trivial idea like "let's build a social network", but quite nontrivial to work out the many technical details associated with bringing that idea to market, from large-scale decisions like using a **symmetric** or **asymmetric** social network by default to small-scale details like how a "Poke" works. It's also extremely nontrivial to actually turn a popular technology into a profitable business; just ask Second Life or Digg.

And that's the reason this phrase is repeated constantly on Hacker News: "It's not the idea, it's the execution" is an excellent reminder, sort of a mantra, a whip for them to self-flagellate themselves into a state of focus. It's also useful for startup novices or dreamers to hear it at least once, as novices tend to assign far too much importance to patents or to seemingly brilliant ideas without working prototypes. As a rule of thumb, early on in a company a patent is only useful as a sort of formal chit to show an investor that you have some kind of defensible technology, and then too a **provisional patent** is fine. And as another rule of thumb, if someone can steal your idea by simply hearing about it, you probably don't have something yet that is truly defensible. Compare *I have an idea for a social network for pet owners* to *I've developed a low-cost way to launch objects into space*.

The net of it is that you can usually be reasonably open about discussing your ideas with people who can give a good bounce, who can understand the idea rapidly and throw

¹Facebook's backend is highly nontrivial ([1](#), [2](#), [3](#)) and comparable only to Google. Even with regards to the UI, just as it is far easier to verify a solution than to arrive at one, it is much easier to copy a UI than to invent it in the first place (especially given the knowledge that a billion people found it congenial).

it back with modifications. The vast majority of people are busy with their own things, or aren't uncreative sorts who are both extremely good at execution and just waiting around to steal your idea. The one exception is [Hacker News](#), and perhaps the startup community more generally. If you have a pure internet startup idea, you should think carefully about whether you want to promote it there before you've advanced it to at least a prototype stage. Unless your customers are developers, you need to raise funding imminently, or you want to recruit developers, you don't necessarily need to seek validation from the startup press and community.

That's one of the reasons that the execution is prized over the idea: people will copy the idea as soon as the market is proved out. No matter how novel, once your idea starts making money the clones will come out of the woodwork. The reason is that it's always easier to [check a solution](#) than to find a solution in the first place. *And a product selling like hotcakes is the ultimate demonstration that you have found a solution to a market need.* Now that you've done the hard work, everyone is going to copy that solution while trying to put their own unique spin on it. So you're going to need excellent execution to survive, because sometimes these "clones" far surpass the originals...as exemplified by Google's victory over AltaVista (and every other search engine) and Facebook's victory over Myspace and Friendster (and every other social network). This is why it sometimes makes sense to keep your head down and grow once you've hit [product-market fit](#) without alerting competitors to the scale of your market opportunity.

Market

There's also a third view. From an investor's standpoint, one can roughly substitute "product" for "idea" and "team" for "execution". That is, given a good idea for a product, a strong team will execute on that idea and push through all the details (financial, regulatory, technical, legal) to get that product to market. With this mapping, the new-new conventional wisdom is that it's neither the product/idea or the team/execution but rather the market. Here's [Marc Andreessen](#) on the topic:

Personally, I'll take the third position – I'll assert that market is the most important factor in a startup's success or failure.

Why?

In a great market – a market with lots of real potential customers – the market pulls product out of the startup. The market needs to be fulfilled and the market will be fulfilled, by the first viable product that comes along. The product doesn't need to be great; it just has to basically work. And, the market doesn't care how good the team is, as long as the team can produce that viable product.

So which is most important: product/idea, execution/team, or market?

One answer is that for a startup veteran or someone who understands any [machine learning](#), the debate is somewhat moot. All these factors (idea, execution, market, team, product) are important to varying degrees; the main point of saying "it's the execution, not the idea" is to disabuse a novice of the idea that patents matter much at inception, or that a brilliant idea on a napkin² has much value. You can think of these things on a continuum. Ideas range

²One argument is that a novel equation or figure is an exception, such as Ramanujan's notebooks or the Feynman diagrams. Something like that truly is valuable, more in the sense of *I've developed a low-cost way*

in quality from “a social network for dogs” to “Maxwell’s equations”, and execution ranges in quality from “I’ll start a company some day” to “sold or IPO’d company”. As important as the execution is, without a clear vision of where you want your company to go you will never come up with an idea that others want to copy. So how do you come up with a good idea?

The Idea Maze

One answer is that a good founder doesn’t just have an idea, s/he has a bird’s eye view of the *idea maze*. Most of the time, end-users only see the solid path through the maze taken by one company. They don’t see the paths not taken by that company, and certainly don’t think much about all the dead companies that fell into various pits before reaching the customer.

The maze is a reasonably good analogy (Figure 1). Sometimes there are pits you just can’t cross. Sometimes you can get past a particular minotaur/enter a new market, but only after you’ve gained treasure in another area of the maze (Google going after [email](#) after it made money in search). Sometimes the maze itself shifts over time, and new doors open as technologies arrive ([Pandora on the iPhone](#)). Sometimes there are pits that are uncrossable for you, but are crossable by another ([Webvan failed](#), but [Amazon](#), [Walmart](#), and [Safeway](#) have the distribution muscle to succeed). And sometimes there are pitfalls that are only apparent when one company has reached scale, problems which require entering the maze at the very beginning with a new weapon (e.g. Google’s Pagerank was inspired in part by Alta Vista’s [problems](#) at scale, problems that were not apparent in [1991](#)).

A good founder is thus capable of anticipating which turns lead to treasure and which lead to certain death. A bad founder is just running to the entrance of (say) the “movies/music/filesharing/P2P” maze or the “photosharing” maze without any sense for the history of the industry, the players in the maze, the [casualties](#) of the past, and the technologies that are likely to move walls and change assumptions (Figure 1).

In other words: a good idea means a bird’s eye view of the *idea maze*, understanding all the permutations of the idea and the branching of the decision tree, gaming things out to the end of each scenario. Anyone can point out the entrance to the maze, but few can think through all the branches. If you can verbally and then graphically diagram a complex decision tree with many alternatives, explaining why your particular plan to navigate the maze is superior to the ten past companies that fell into pits and twenty current competitors lost in the maze, you’ll have gone a long way towards proving that you actually have a good idea that others did not and do not have. This is where the historical perspective and market research is key; a strong new plan for navigating the idea maze usually requires an obsession with the market, a unique insight from deep thought that others did not see, a [hidden door](#). You can often distinguish these good ideas because they require the explanation of an arcane acronym or regulation like [KYC](#) (Paypal) or [LDT](#) (Genomic Health) or [OTARD](#) (Aereo).

The Execution Mindset

We’ve spoken about what good ideas are. What does good execution involve, in concrete terms? Briefly speaking, the execution mindset means doing the next thing on the todo list at all times and rewriting the list every day and week in response to progress. This is easy to say, extremely hard to do. It means saying no to other people, saying no to distractions, saying no to fun, and exerting all your waking hours on the task at hand. The execution

to launch objects into space than I have an idea for a social network for pet owners.

The Idea Maze

A "good idea" is a detailed path through the maze. Why does your path lead to treasure: competitor oversight, new technology , or something else?

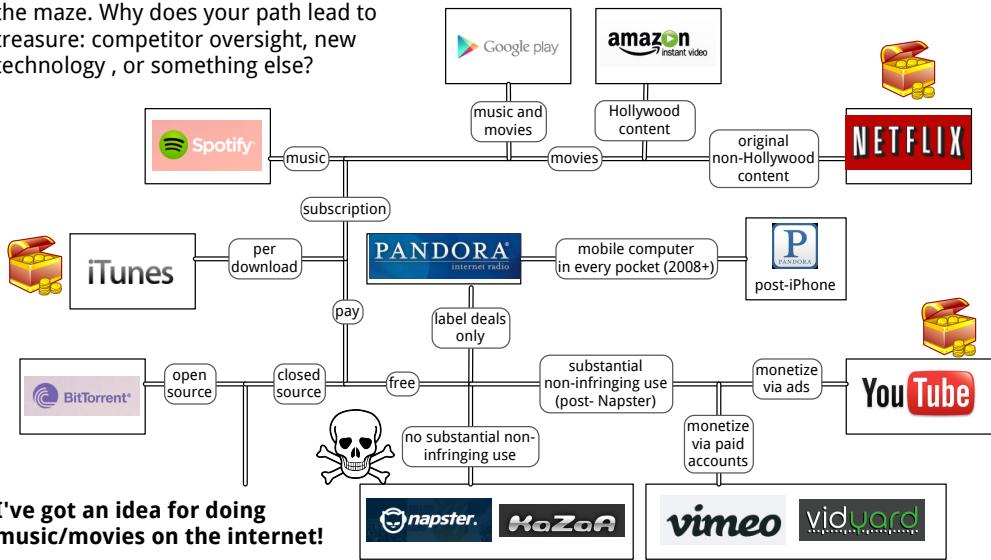


Figure 1: Visualizing the idea maze. A good idea is not just the one sentence which gains you entrance to the maze. It's the collection of smaller tweaks that defines a path through the maze, along with explanations of why this particular combination will work while others nearby have failed. Note that this requires an extremely detailed understanding of your current and potential competitors; if you believe you don't have any competitors, think again until you've found the most obvious adjacent market. This exercise is of crucial importance; as Larry Ellison said, "Choose your competitors carefully, as you will become a lot like them."

mindset is thus about running the maze rapidly. Think of each task on the list as akin to exploring a turn of the maze. The most important tasks are those that get you to the maze exit, or at least a treasure chest with some powerups.

In terms of execution heuristics, perhaps the best is Thiel's [one thing](#), which means that everyone in the company should at all times know what their one thing is, and others should know that as well. Marc Andreessen's [anti-todo](#) list is also worthy of mention: periodically writing down what you just did and then crossing it off. Even if you get off track, this gives you a sense of what you are working on and your progress to date.

Implementing “Thiel’s One Thing” with Gmail chat status. Good tools can help with execution. For personal tasks, [emacs orgmode](#) is highly recommended as it's completely keyboard driven, works offline, scales from a simple todo list to a full blown document, and integrates arbitrary code snippets. For company-wide tasks, a combination of Github Issues and Gmail Chat status messages can be used to broadcast the link of the issue you're currently working on.

If there's something that's worth nagging people about, it's keeping their Gchat status up to date with their latest Github issue in this fashion. It constantly refocuses you on your one thing, and allows the entire company to know what that is without constant interrupts. Periodically, you too can scan what others are working on without disrupting their flow.

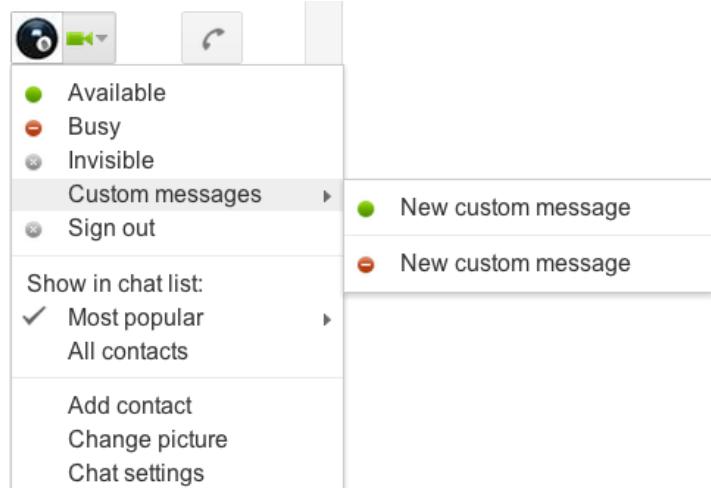


Figure 2: To implement the “one thing” heuristic with Gmail status and Github issues, set up a new custom message.

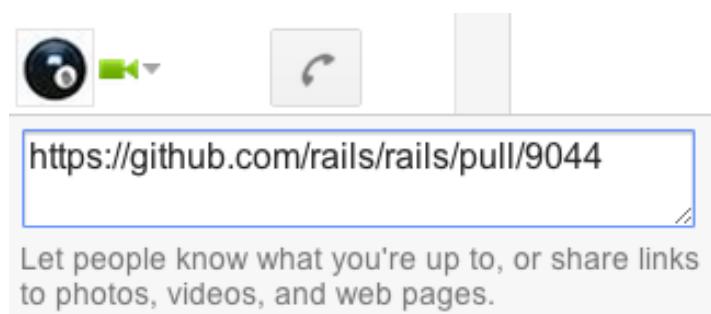


Figure 3: Now anyone can click to see the one thing you are working on, and can comment in your ticket. The only thing you are doing is what is in your status.

What Kind of Business Do you Want to Build?

Startups vs. Small Businesses

It's useful at the very outset to distinguish between startups and small businesses (see also [here](#)). A startup is about growth: it is a play for world domination, an attempt to build a business with global reach and billions in annual revenue. Examples include Google, Facebook, Square, AirBnB, and so on. Startups usually involve new technology and unproven business models. A small business, by contrast, does not have ambitions of world domination. It is usually geared towards a particular geographical area or limited market where it has some degree of monopoly through virtue of sheer physical presence. Examples include pizza parlors, laundromats, or a neighborhood cafe.

With these definitions you can start to see why startups are associated with the internet and small businesses with physical storefronts. It's easier to scale something virtual from 1 to 1 billion users. Having said that, there are virtual operations that make a virtue out of being "lifestyle businesses" (e.g. 37Signals) and physical operations that yield to none in their ambition for scale and world domination (e.g. Starbucks, McDonalds).

Another important distinction is that small businesses must usually generate profits right away, while startups often go deep into the red for quite a while before then sharply turning into the black (in a good scenario) or going bankrupt (in a bad one³). Ideally one accumulates the capital to start either a small business or a startup through after-tax savings coupled with a willingness to endure extremely low or zero personal income during the early non-revenue-generating period of the small business/startup.

But soon after inception one encounters a crossroads: should you take outside capital from VCs to try to grow the business more rapidly and shoot for the moon, or should you rely on steady organic growth? Taking venture capital is much like strapping a jetpack to a man who's been running at a nice trot. It could lead to a [terrible wipeout](#) or a flight to the moon. The most important consideration in this decision is your own ambition and personal utility function: can you tolerate the failure of your business? Because if you can't go to zero, you shouldn't try to go to infinity ([1](#), [2](#), [3](#)).

Startups Must Exhibit Economies of Scale

So let's say that you do want to go to the moon rather than build a lifestyle business. Your next step is to do a simple calculation to determine whether a given business is capable of getting there, namely whether it exhibits an economy of scale.

Suppose that we have a startup with the following property: it sells each unit for \$1000, but the per-unit cost drops as more units are sold, exhibiting an economy of scale, as shown in Table 2. A table of this kind can arise from an upfront cost of \$50,000 for software development to handle the first 100 orders $[(\$1200-\$700) * 100]$, and then another \$247,500 $[(\$975-\$700)*900]$ in fixed costs for design/manufacturing revisions to support more than 1000 customers. After those two fixed expenditures, the startup is only paying per-unit costs, like the amortized cost of customer service and the cost of materials.

This simple calculation illustrates many things about the startup world. First, one can see immediately how important it is to shift per-unit costs into fixed costs (e.g. via software),

³Note though that even a money-losing company is an asset, though: under the US tax code, losses in previous years can be counted against profits in current years, so acquiring a money-losing business for a song can be an excellent way for a large company to shield some profits from taxes (see [deferred tax assets](#)).

and why seemingly costly up-front software development can pay off in the long run. Second, one can determine how much capital is needed before the business breaks even; this is one of several calculations that a venture capitalist will want to see before investing nontrivial money into the business. (Note that your low point is usually much lower than in your perfect model!) Third, one realizes how important pricing is; insofar as you are not constrained by competition, you really do want to charge the highest possible price at the beginning in order to get into the black as soon as possible. A seemingly insignificant change from \$1000 up to \$1200 would completely change the economics of this business and make it unnecessary to take on outside capital. Moreover, free or **heavily discounted** customers generally don't value the product and are counterintuitively the most troublesome; paying customers are surprisingly more tolerant of bugs as they feel like they're invested in the item. Fourth, one starts to understand why it is so incredibly difficult to achieve \$199 or \$99 price points without massive scale. A real product has dozens if not hundreds of cost components, each of which has their own economy-of-scale function, and each of which needs to be *individually* driven to the floor (via robotics, supply chain optimizations, negotiation, etc.) in order to decrease the overall price of the product. It's hard to make a profit!

Table 2: A product that exhibits an economy of scale. The cost function is plotted in Figure 4.

Number of units	Cost of production per unit	Revenue per unit
$0 \leq n \leq 100$	\$1200	\$1000
$101 \leq n \leq 1000$	\$975	\$1000
$1001 \leq n$	\$700	\$1000

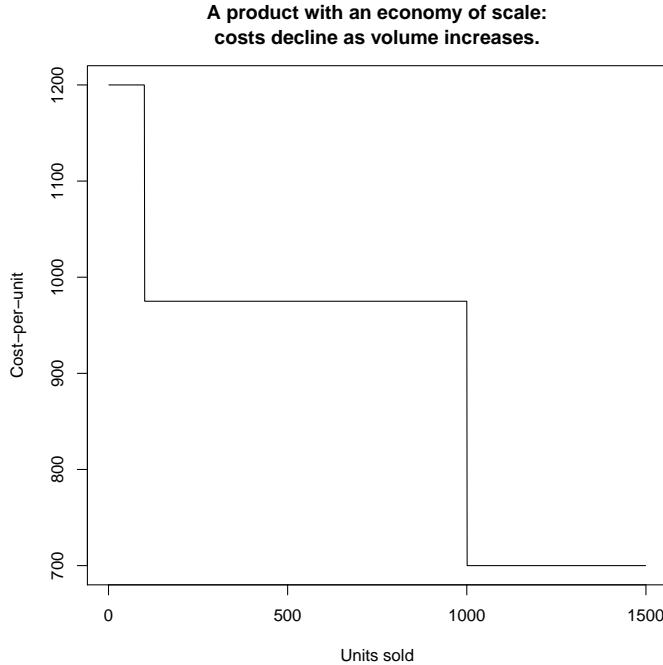


Figure 4: A product with an economy of scale: costs decline with volume.

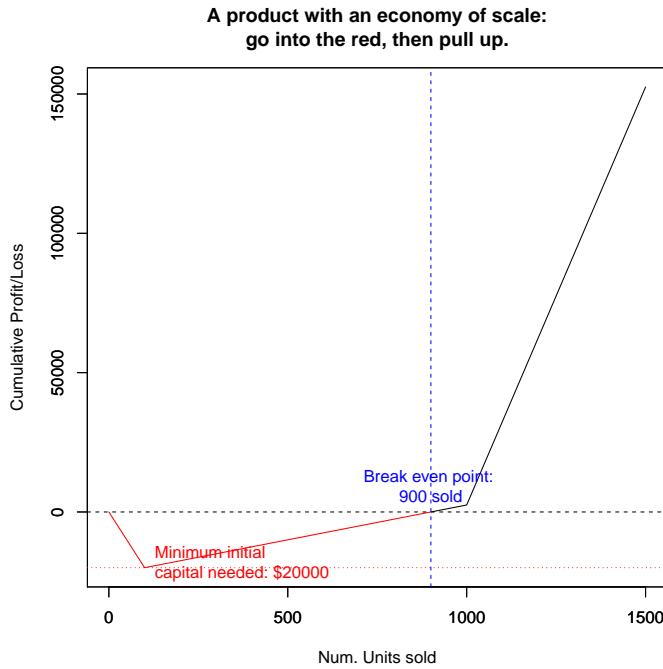


Figure 5: A product with an economy of scale: initially one goes into the red, then pulls up.

Startups Must Pursue Large Markets

Even if you can build a product with an economy of scale, you need to ensure that it serves a large market. The annual *market size* is the total number of people who will buy the product per year multiplied by the price point. To get to a billion dollars in annual revenue (\$1B), you need either a high price point or a large number of customers. Here are a few different ways to achieve that magic \$1B figure in different industries:

- *Sell product at \$1 to 1 billion:* Coca Cola (cans of soda)
- *\$10 to 100 million:* Johnson and Johnson (household products)
- *\$100 to 10 million:* Blizzard (World of Warcraft)
- *\$1,000 to 1 million:* Lenovo (laptops)
- *\$10,000 to 100,000:* Toyota (cars)
- *\$100,000 to 10,000:* Oracle (enterprise software)
- *\$1,000,000 to 1,000:* Countrywide (high-end mortgages)

Now, some of these markets are actually much larger than \$1B. There are many more than 100,000 customers for \$10,000 cars in the world, for example; the number is more like 100,000,000 [annual customers](#) for \$10,000 cars. So the annual automobile market for new cars is actually in the ballpark of \$1T, not \$1B. And that's why Google is going after [self-driving cars](#): it's one of the few things that is a substantially bigger market than even search advertising.

Note in particular that low price points require incredible levels of automation and industrial efficiency to make reasonable profits. You can't tolerate many returns or lawsuits for each \$1 can of Coke. Relatedly, high price points subsidize sales; it is not 100,000 as much sales time to sell a house as it is to sell a can of Coke, but it is 100,000 as much revenue.

Do Market Sizing Calculations Early and Often

With this basic grounding in market size, take a look at Figure 6. This simple schematic is what many doctorates wish they'd been shown on the first day of graduate school. As background, your modal PhD student works on a project which is technically challenging and highly prestigious within their advisor's subculture. After six years, if they are entrepreneurially inclined, they might think about commercializing their technology. It is at this point that all too many do their *very first* back-of-the-envelope calculation on a napkin about the possible market size of their work. And often that calculation does not go well.

Instead, an incredible amount of time and energy could have been saved if at the beginning the student had simply enumerated every project s/he found equivalently scientifically interesting, and then rank-ordered said projects by their market size. With all else held equal, if you have committed to spending several years of your life on a PhD, you want to pick the project with the largest market potential.

Among other things, market size determines how much money you can raise, which in turn determines how many employees you can support. As a back of the envelope, suppose that the average Bay Area startup employee costs \$100,000 "all in", with this figure including not just salary but health insurance, parking spaces, computer equipment, and the like. Suppose further that you think you need just five employees for three years to come up with the cure for a rare disease. Then to support just five employees per year is \$500,000 per year, not including any other business expenses. But if your market size is only \$50M, then you have a problem, because it is going to be very difficult to find someone who will invest \$1.5M for three years of R&D to pursue such a small market. Among other things, you won't capture the entire market, but only a portion. And if it's going to take three years, that \$1.5M could go towards something that has a chance at a \$1B+ market.

The best market size estimates are both surprising and convincing. To be surprising is the art of the presentation. But to be convincing, you want to [estimate](#) your market size in at least two different ways. First, use [Fermi estimates](#) to determine the number of people who will buy your product (top-down market sizing). This will require knowledge of general stats like 300M Americans, 6B world population, 6M US businesses, as well as domain-specific stats like 4M pregnancies (see also Figures 7 and 8). Second, use SEC filings of comparable companies in the industry to get empirical revenue figures and sum these up (bottom-up market sizing); we did this in HW2 with market-research.js. The latter is generally more reliable, though you must be sure that you are not drawing your boundaries too broadly or optimistically ("if we only get 1% of China..."). One of the most convincing things you can do with a bottoms-up estimate is to directly link or screenshot an invoice with a high price point; for example, one has no idea how high the price points in the [recruiting](#) or [PR](#) markets are until you actually encounter them.

Market size vs. novelty

If you are choosing between two projects of equal scientific interest, pick the one with the larger market size. And do that calculation up front, not at the end of your six year PhD program.

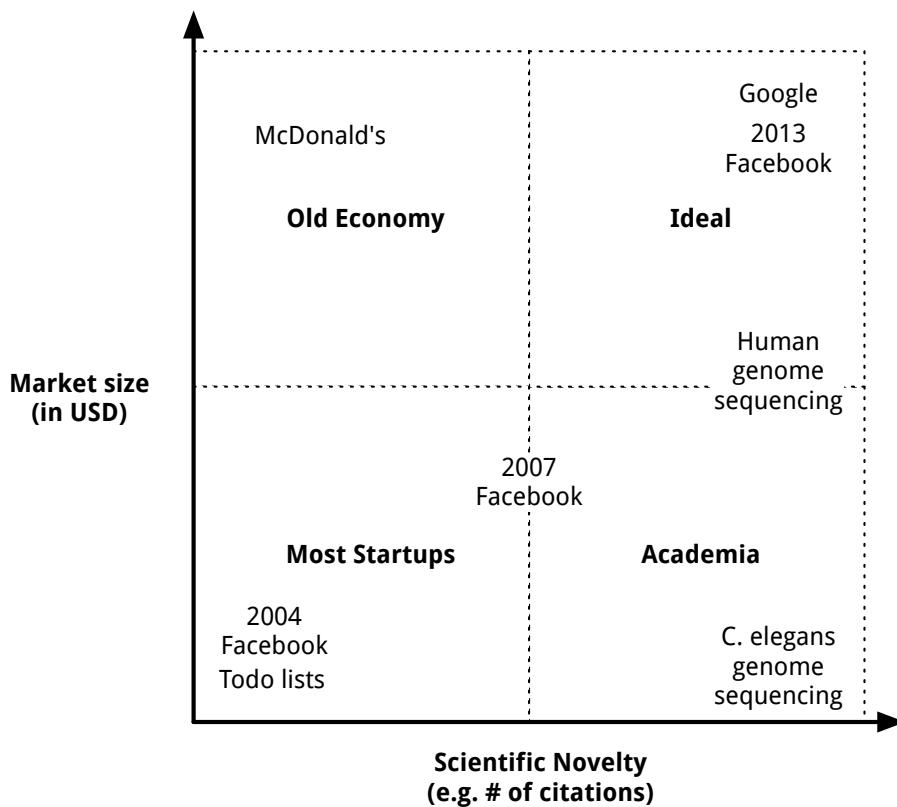


Figure 6: Qualitative graph of market size (in \$) vs. scientific novelty (in citations or impact factor).

Market Research

So we now understand that a good idea means a bird's eye view of the *idea maze* and that the execution mindset means rapidly navigating the idea maze. We also see how market size drives not just the relevance of the company, but also controls downstream parameters like the quantity of money that can be raised and hence the number of employees.

Tools for market research

Let's now get down to brass tacks and talk about tools for market research. Suppose that you've identified a possible market and want to drill down. For example, suppose your hypothesis based on recent news stories is that the US dollar will eventually be replaced by the [Chinese yuan](#) as the world's [reserve currency](#), and you believe a service that would make it easier for people to move their bank accounts from USD to RMB would be of value. How would you validate this hypothesis?

1. First, you'd want to have some news coverage or research papers to establish the overall frame; these will be display pieces in any pitch to investors. Going to Google Books for the history and reading [SEC filings](#) and Wikipedia can be extremely helpful.
2. Next, you want to do a back of the envelope estimate of market size, Googling as necessary for any statistics.
3. Then you want to further validate this market with some modern tools, including [Google's Keyword Planner](#) and [Facebook's Advertiser Tools](#). (Figures 7-8).
4. If that proves fruitful, you should develop a simple landing page by using a service like [Launchrock](#), photos from [iStockphoto](#), and icons from iconfinder. You may also want to read up on some [basic SEO](#). We'll mostly focus on the crowdfunder, but Launchrock can be a complement to that approach. If your users need to see flows to understand the product, you will want to create some wireframes (see Figure 10).
5. Finally, you want to allocate a small [Google Adwords](#) or [Facebook Ads](#) budget to test out the market, and see how many conversions you get.

The overall concept here is to establish that a market exists *before* building a so-called [minimum viable product](#) (MVP). This process is related to some of the ideas from the [Lean Startup](#) school of thinking. Now, the ideas from this school can be taken too far... as Henry Ford famously noted, if he had asked his customers what they wanted, they would have asked for a faster horse. But most people new to startups aren't in danger of talking to too *many* customers.

The screenshot shows the Facebook Audience Insights interface. At the top, there's a large blue "facebook" logo. Below it, the title "Choose Your Audience" is displayed. On the left, there are several targeting parameters:

- Location:** United States (selected), with options for Country, State/Province, City, and Zip Code.
- Age:** 13-34 (selected), with a dropdown for "No max" and a checked checkbox for "Require exact age match".
- Gender:** All (selected), with options for Men and Women.
- Precise Interests:** A search bar containing "#Kickstarter". Below it is a list of broad categories: Activities, Business/Technology, Events, Family Status, Games, Interests, Market, Mobile Users (All), and Mobile Users (Android). Under "Activities", there's a list of specific interests: Console Gaming, Cooking, Dancing, DIY/Crafts, Event Planning, Fast Food Diners/QSR, Food & Dining, Frequent Casual Diner, and Gaming (Social/Online).
- Connections:** Anyone (selected), with an option for Advanced connection targeting.
- Friends of Connections:** A search bar for targeting people whose friends are connected to a specific page, app, or event.

On the right side, there's a summary section titled "How to Choose an Audience for Your Ad". It shows the total audience size as **1,060,000** people, with two sub-points: "who live in the United States" and "who like #Kickstarter". Below this, a "Suggested Bid" section shows a range of **\$0.18-\$1.26 USD**.

At the bottom of the targeting panel, there's a link to "See Advanced Targeting Options".

Below the targeting panel, there's a summary section with the heading "facebook.com/advertising" and the text: "FB Ads: Outstanding census tool for doing realtime census of people interested in a specific idea. Fantastic for bottoms-up market research."

Figure 7: Facebook's realtime census tool is one of their most underappreciated features. Go to facebook.com/advertising and log in. Then begin creating an ad, and you'll see the realtime census tool depicted above. You can use this to complement the Google tools and do some incredible realtime market research. For example, you can find the number of men ages 18-34 in Oregon who have an interest in snowboarding. These kinds of numbers let you get quick upper bounds on market sizes.

The screenshot shows the Google Keyword Planner interface. On the left, there's a sidebar with options like 'Search for keyword and ad group ideas', 'Your landing page', 'Your product or service', 'Targeting' (with dropdowns for location, language, and Google), and 'Customize your search' (with filters for keyword filters and CPC). A prominent blue button at the bottom says 'Get Ideas'. On the right, the main area is titled 'Your product or service' with the input 'crowdfunding'. Below this, there are two tabs: 'Ad group ideas' (selected) and 'Keyword ideas'. A table follows, with columns for 'Ad group (by relevance)', 'Keywords', 'Avg. monthly searches', and 'Competition'. The table contains the following data:

Ad group (by relevance)	Keywords	Avg. monthly searches	Competition
Kickstarter (8)	kickstarter, kickstarter cl...	1,001,390	Low
Keywords like: Crowdfun...	crowdfunding, crowdfoun...	384,190	Medium
Fundraising Ideas (65)	fundraising ideas for kids...	205,660	High
Fundraising (112)	fundraising activities, ea...	155,710	Medium
Business Loans (5)	business loans, small bu...	93,180	High
Crowdsourcing (14)	crowdsourcing, crowdso...	80,060	Low
Crowd (59)	crowd funding, crowd fun...	39,650	Medium

adwords.google.com/ko/KeywordPlanner
Google Adwords' Keyword Planner gives you bottoms-up statistics on search queries. This is more suitable for assessing purchase intent than determining the number of individuals with a given interest.

Figure 8: You can use Google's Keyword Planner at adwords.google.com/ko/KeywordPlanner to help estimate market size. Simply figure out some keywords related to your market, plug them in to get daily search volumes, and estimate what fraction of those would actually convert and at what price point. Use a few different values to get rough lower and upper bounds for the market. Then repeat with a few other related keyword sets to triangulate on a result.

A framework for determining product tiers

Once you've got your overall market that you're targeting, you'll need to come up with a product idea. For now let's call that your moment of insight and creativity. If you have that product idea, though, in what order do you build features? First, read about [Kickstarter product tiers](#) and [trends](#). The heuristic price points in [this post](#) are a good start, but we can supplement this with a little bit of theory, as shown in Figure 9.

The basic idea is that the ultimate market research is a table with 7 billion rows (one for each person), N columns on their attributes (e.g. location, profession) and K columns (one for each product version), with each entry giving the amount that person will pay for those features. Now, of course you won't be able to sample all 7 billion, but you can sample a few hundred (e.g. with [Google Consumer Research Surveys](#) or [Launchrock landing pages](#)), and this gives you a conceptual framework for how to set up your product tiers.

Continuing with the conceptual example, in Figure 9 the CEO with PersonID 3 (i.e. the 3rd person out of 7 billion people) wants your product so much that s/he will pay up to \$1000 for the version 1 with features 1 and 2. The vast majority of people will pay \$0 no matter how many features you add. Histograms on these columns give the range of pricing that the market will tolerate, and allow you to estimate the amount of revenue you will make on each version (see in particular Spolsky on [price segmentation](#)). In particular, you want to confirm that you will make enough money on version 1 to pay for version 2. If this is not the case, you should rearrange the order of features until it is true, at least on paper.

If you are seriously considering starting a business, paying \$250 for a demographically targeted [survey](#) with 500 responses will save you much more than that in the long run in terms of time/energy expenditure, at least if you're in the US. Even spending \$1000 to seriously evaluate four ideas will be well worth it in terms of the time/energy savings (though you should do one survey and review the results before you commit to doing more). If you genuinely can't afford this, ask 10-20 of your friends how much they'd pay for different version of your product, or run a poll on Facebook. You will find that early feedback on pricing and features - even biased feedback - is greatly superior to none at all.

Product Tier Selection: A Conceptual Framework

Once you've settled on a market and a broad product idea, you need to think about exactly what your product versions will be and what features you will prioritize in each version. Here's a simple conceptual framework to guide that process, ensuring that you make enough revenue from each step to fund the development of the next stage.

Step 1: What is the maximum price each demographic would pay for each version of your product? You can't assess all 7e9, but you can survey a subset.

PersonID	Job	Employer	Age	Gender	Version 1	Version 2	Version 3
1	Engineer	Exxon	32	M	\$100	\$200	\$200
2	Mechanic	Self	23	F	\$0	\$10	\$10
3	CEO	Startup Co	39	M	\$1000	\$1500	\$2000
4	Barista	Starbucks	22	M	\$0	\$0	\$0
...
7e9	Scientist	NIH	27	F	\$0	\$0	\$0

Step 2: What features are associated with each version?

	Feature 1	Feature 2	Feature 3	Feature 4
Version 1	Y	Y	N	N
Version 2	Y	Y	Y	N
Version 3	Y	Y	Y	Y
...

Step 5: Calculate market size
Rough global market size for v1 with perfect segmentation

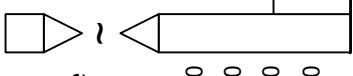
$$\begin{aligned} & (100,000 * \$1000 + \\ & (10,000 * (\$1100 + \$1200)) + \\ & (1000 * (\$1300 + \dots + \$2000)) \\ & = \$136.2M \end{aligned}$$

Rough global market size for v1 priced at \$1000: \$128M

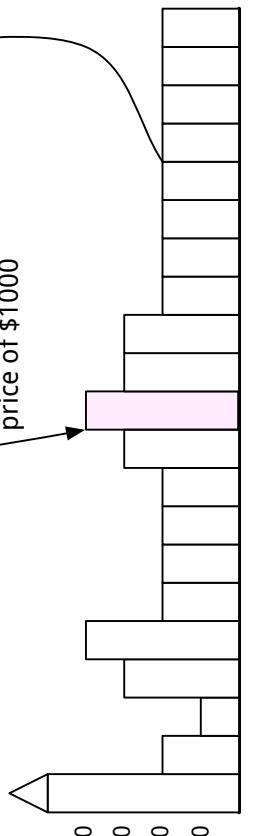
Is this greater than the cost of v1's features? In this case, yes.

Step 4: From your survey, what does the histogram of maximum prices ("reserve prices") look like for v1, and what's the estimated revenue?

6.9e9 (most will never buy)



of people who would buy at this price ~



Step 3: What is the estimated cost/time to build each feature?

	Cost	Time
Feature 1	\$1,000,000	50 days
Feature 2	\$50,000	100 days
Feature 3	\$500,000	200 days
...



Version 1: Reserve Price

Wireframing, Copywriting, and Design

Once you feel that you have a market, you'll want to design a mockup of your product's website, also known as a wireframe. For the purposes of this class, we'll presume two things: first, that every product in the class will have a web component. Second, that the product itself will be initially marketed through your own crowdfunding website, but will ultimately have its own dedicated website. Indeed, this crowdfunding website can ultimately morph into the product website over time.

Wireframing

There are many excellent wireframing tools, and we've picked out four in particular that are of interest Figure 10. For editing on the computer, [Omnigraffle Pro](#) is very good for offline editing, [Lucidchart](#) is useful for collaborative wireframing, and [Jetstrap](#) makes it easy to convert a final design to Twitter Bootstrap. For converting offline pen and paper mockups, [POP \(popapp.in\)](#) is really worth trying out. Try these tools out and find one you like; your main goal is to produce sitemaps and detailed user flows that look [something like this](#).

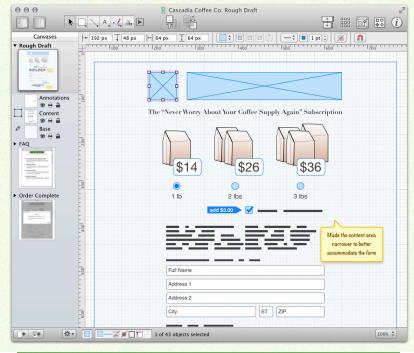
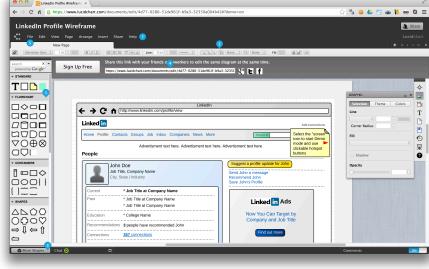
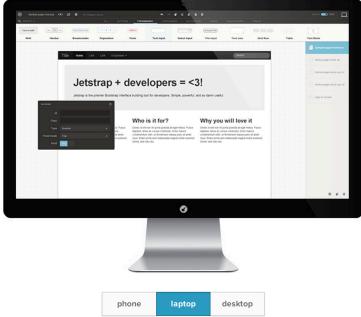
 <p>OmniGraffle for Mac omnigroup.com/products/omnigraffle Advantage: Offline support, many stencils at Graffletopia, and excellent keyboard shortcuts</p>	 <p>Lucidchart lucidchart.com Advantage: Supports multiplayer collaboration and works with Google Drive</p>
 <p>jetstrap.com Advantage: Directly export to HTML/CSS/JS and mockup with Bootstrap objects</p>	 <p>popapp.in Advantage: Prototype with pen and paper and easily make clickable workflows</p>

Figure 10: A survey of four useful wireframing tools. Pick the one that works best with your style.

Copywriting

Here are some high level copywriting principles.

- *Homepage message*: If it's not a homepage message, it's not a message. Don't expect visitors to figure out what the product is about on the third page in; your enemy is the back button. That said, only spend energy on the homepage if it's a significant source of customers. If it isn't (e.g. if your sales come through a salesforce), you can leave the homepage alone for a surprisingly long time.
- *Work backwards from the press release*: Amazon writes the press release before they build the product. In doing this, you will find yourself figuring out what features are news and which ones are noise. Take a look at their [press releases](#) to get a sense for tone.
- *Someone is wrong on the internet*: Get a friend, have them pull up as many competitor webpages as possible, and you will quickly find yourself explaining to them why those companies are terrible and why yours is different. This is valuable. Write down your immediate gut reaction as to why your product is indeed different. Put this into a feature matrix ([example](#)) and then determine whether your new features really differentiate your product substantially.
- *Simple and factual*: Do your best to turn vague features ("PageRank provides better search quality") into concrete facts and statistics ("X% of searchers who used both Blekko and Google preferred Blekko. Find out why."). Use [Oracle style marketing](#): less is more, hammering one point [above the fold](#) and keeping the rest in the secondary copy.
- *Call to action*: You want someone to do something when they come to your page, like buy something or sign up on an email form. Make sure you reiterate this "call to action" many times. Make the call to action button cartoonishly large and in a different font.

You can put these principles to work with your crowdfunder.

Design

Your first goal with the wireframe should be functionality and semantic meaning: what does the product do? Your second goal is the marketing copy: how do we explain what the product does in words? Your final goal is the design: how do we make the site and product look beautiful and function beautifully? Let's go through some high-level design principles.

- *Vector and Raster Graphics*: The first thing to understand about design is the distinction between [vector](#) and [raster](#) graphics. Whenever possible, you want to work with vector objects as they will scale well and are amenable to later manipulation. Raster images are unavoidable if you truly need photos on your site, but you should strive to minimize them for your MVP.
- *Alignment, Repetition, Contrast, Proximity*: The [Non-Designer's Design Book](#) by Robin Williams is reasonable as a next step, with the four principles summarized [here](#). You can see what these four principles look like in practice: [Alignment](#), [Repetition](#), [Contrast](#), [Proximity](#). A good strategy is to look at the examples, try to internalize the lessons, create your design without thinking too much about them, and then re-apply them over the design at the end.

- *Fonts and Icons:* In terms of getting something reasonably nice together quickly, you should make heavy use of fonts. Because fonts are vector graphics specified by mathematical equations, they are infinitely manipulable with CSS transforms built into every browser (which we will cover next week). The [FontAwesome](#) library also gives scalable vector icons, which are often useful for quick mockups when enlarged in size. Even if not the perfect image, the fact that these complex images can be generated in a single keystroke and rotated/colored with a few more make them quite good for quick proofs-of-concept. With a bit of practice, one can get to a reasonably good looking prototype with pure fonts and Unicode characters.
- *Stock Photos, Videos, Animations:* Use iStockphoto as a visual companion to the Key-word Tool External. Often the key to a good presentation or webpage is to put an abstract concept into visual form. Stock photographers have actually thought about these issues; here's an example for [launching a business](#). One point: don't use stock photos in which people are looking at the camera. That's what makes a stock photo look cheap. A good signal of a lower quality site is that it has inconsistent padding, doesn't use [https](https://) properly for checkout, and/or uses stock photos of this kind. Again, this is [bad](#), while this is [passable](#). You might also consider reshooting a stock photo with your product so that it looks custom.
- *Use Bootstrap, Themeforest, 99Designs, Dribbble:* When it comes to turning your wireframes into HTML/CSS/JS, don't reinvent the wheel. Design is information and is becoming commoditized. [Bootstrap](#) is a free CSS framework we'll be using through the course. Jetstrap, mentioned previously, is a wireframing tool that exports to functional HTML/CSS with Bootstrap. [Themeforest](#) is a set of slightly more expensive (\$9-12) templates. [99Designs](#) is a few hundred to a few thousand bucks for a logo or webpage revision. And [Dribbble](#) is an excellent resource for finding a contract or full time designer.

An introduction to HTML/CSS/JS

A webpage is a program

It's useful to think of a webpage as similar to a C program from Stanford's [CS106B](#). Just like you use [gcc](#) to [compile](#) C programs at the command line, when you navigate to a page in a web browser it performs a kind of realtime compilation step called [rendering](#). The input is raw text like [the left side of this page](#) and the output is the clickable GUI interface in your browser.

Once you start thinking of webpages as akin to programs, you begin to realize the level of sophistication of modern web browsers. Not only do browsers need to compile a fairly complicated language (HTML/CSS/JS) almost instantly to create a user-friendly clickable interface, they have to deal with the fact that you as the end-user are essentially *installing* a program every single time you view a page. They also need to deal robustly with noisy network connections, bizarre character encodings, legacy code dating back to the [early 1990s](#), security considerations of all varieties, and many other complexities that most programs can safely ignore. It's thus no exaggeration to say that a web browser is the most complex piece of code on a computer, comparable only to a compiler or operating systems in terms of its complexity. Indeed, Chrome itself is now the core of an operating system (namely [Chrome OS](#)).

From HTTP Request to Rendered Page

Emil Stenstrom put together an excellent high-level overview of the [rendering process](#) in the context of a modern web application; let's go through an edited and updated version.

1. You begin by typing a [URL](#) into address bar in your preferred [browser](#) or clicking a link.
2. The browser parses the URL to find the protocol, host, port, and path ([example](#)).
3. If HTTP was specified, it forms a [HTTP request](#).
4. To reach the host, it first needs to translate the human readable host into an [IP address](#), and it does this by doing a [DNS lookup](#) on the host ([video](#)).
5. Then a [socket](#) needs to be opened from the user's computer to that IP address, on the [port](#) specified (most often [port 80](#) for HTTP).
6. When a [network connection](#) is open, the HTTP request is sent to the host. The details of this connection are specified in the [7-layer OSI model](#) (see [examples](#)).
7. The host forwards the request to the [server software](#) (most often Apache) configured to listen on the specified port.

8. The server inspects the request (most often only the path), and the subsequent behavior depends on the type of site.
 - For [static content](#):
 - The HTTP response does not change as a function of the user, time of day, geographic location, or other parameters (such as [HTTP Request headers](#)).
 - In this case a fast static web server like [nginx](#) can rapidly serve up the same HTML/CSS/JS and binary files (jpg, mp4, etc.) to each visitor.
 - Academic webpages like <http://startup.stanford.edu> are good examples of static content: the experience is the same for each user and there is no login.
 - For [dynamic content](#):
 - The HTTP response *does* change as a function of the user, time of day, geographical location, or the like.
 - In this case you will usually forward dynamic requests from a web-server like nginx (or [Apache](#)) to a constantly running server-side [daemon](#) (like [mod_wsgi](#) hosting [Django](#) or [node.js](#) behind [nginx](#)), with the static requests intercepted and returned by nginx (or even before via [caching layers](#)).
 - The [server-side web framework](#) you use (such as [Python / Django](#), [Ruby / Rails](#), or [node.js / Express](#)) gets access to the full request, and starts to prepare a HTTP response. A web framework is a collection of related libraries for working with HTTP responses and requests (and other things); for example, here's the Express [request](#) and [response](#) APIs, which allow programmatic manipulation of requests/responses as [Javascript objects](#).
 - To construct the HTTP response a [relational database](#) is often accessed. You can think of a relational database as a set of tables similar to Excel tables, with links between rows (example [database schema](#)).
 - While sometimes raw [SQL](#) is used to access the database, modern web frameworks allow engineers to access data via so-called Object-Relational Mappers (ORMs), such as [sequelize.js](#) (for node.js) or the [Django ORM](#) (for Python). The ORM provides a high-level way of manipulating data within your language after defining some models. (Example [models / instances](#) in sequelize.js).
 - The specific data for the current HTTP request is often obtained via a database search using the ORM ([example](#)), based on [parameters](#) in the path (or data) of the request.
 - The objects created via the ORM are then used to [template](#) an HTML page ([server-side templating](#)), to directly return JSON (for usage in [APIs](#) or [client-side templating](#)), or to otherwise populate the body of the [HTTP Response](#). This body is then conceptually put in an envelope with [HTTP Response headers](#) as metadata labeling that envelope.
 - The web framework then returns the HTTP response back to the browser.
9. The browser receives the response. Assuming for now that the web framework used server-side templating and return HTML, this HTML is parsed. Importantly, the browser must be robust to [broken or misformatted](#) HTML.

10. A [Document Object Model](#) (DOM) tree is built out of the HTML. The DOM is a tree structure representation of a webpage. This is confusing at first as a webpage may look planar rather than hierarchical, but the key is to think of a webpage as composed of chapter headings, subsections, and subsubsections (due to HTML's [ancestry](#) as a descendant of [SGML](#), used for formatting books).
11. All browsers provide a standard programmatic [Javascript API](#) for interacting with the DOM, though today most engineers manipulate the DOM through the cross-browser [JQuery library](#) or higher-level frameworks like [Backbone](#). [Compare and contrast](#) JQuery with the legacy APIs to see why.
12. [New requests](#) are made to the server for each new resource that is found in the HTML source (typically images, style sheets, and JavaScript files). Go back to step 3 and repeat for each resource.
13. [CSS](#) is parsed, and used to annotate each node in the DOM tree with style information on how it should render. CSS controls appearance.
14. [Javascript](#) is parsed and executed, and DOM nodes are moved and style information is updated accordingly. That is, Javascript controls behavior, and the Javascript executed on [page load](#) can be used to move nodes around or change appearance (by updating or setting CSS styles).
15. The browser renders the page on the screen according to the DOM tree and the final style information for each node.
16. You see the webpage and can interact with it by clicking on buttons or submitting forms. Every link you click or form you submit sends another HTTP request to a server, and the process repeats.

To recap, this is a good overview of what happens on the server and then the client (in this case the browser) when you navigate to a URL. Quite a miracle, and that's just what happens when you click a link¹.

Anatomy of a webpage

Let's now drill down a little from the macroscopic context of HTTP requests and responses to get a bit more detail on the components of a webpage: HTML/CSS/JS. If you've ever written a comment on a blog, you're probably familiar with basic HTML tags like `` to bold text

¹It's useful to think in a bit more detail about what happens when you hit a key to start a job on a remote EC2 machine. Strike a key and the capacitance changes instantaneously, sending a signal down the USB cable into the computer. The computer's keyboard driver traps the signal, recognizes the key it's from, and sends it to the operating system for handling. The OS determines which application is active and decides whether the keystroke needs to be routed over an internet connection. If it is routed, a set of bytes is sent down the CAT-5 cable into the wall socket, abstraction masked via the 7-layer model, and sent to a router. The router determines the closest router which is likely to know the IP of your destination, and sends that packet onwards. Eventually the packet arrives at the destination computer, enters the via ethernet cable, is trapped by the remote OS, interpreted as an enter stroke, sent to the shell, sent to the current program, and executed. Data from that remote machine is then sent back over the cable and the routers to your machine, which takes the packet, uses it to update the display, and tells you what happened on that remote box. All of that happens within an imperceptible fraction of a second.

and `<i>` to italicize. But to build a proper webpage you need to understand the division of labor between HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JS (Javascript), abbreviated as HTML/CSS/JS.

If you analogize a webpage to a human, you won't go wrong by thinking of HTML as the skeleton, CSS as the skin and clothing, and Javascript as the dance routine that makes the human move (and possibly change clothing). There are many outstanding books and tutorials in this general area, though be warned that w3schools.com is only so-so despite its incredible PageRank. The WebPlatform.org tutorials on [HTML](#), [CSS](#), and [JS](#) are useful indexes of tutorials. You can also use htmldog.com to go through HTML and CSS, and [Eloquent Javascript](#) for JS.

HTML: Skeleton and Semantics

HTML is a finite set of about 112 [elements](#) (aka tags) that are used to specify the structure of a webpage. Each element has [attributes](#), key-value pairs that specify metadata on the element or modify its appearance in the browser. Here's an example with the `<abbr>` tag for an abbreviation in which four attributes are explicitly specified:

```
1 <abbr id="sql-abbrev" class="abrevs" style="color:purple;"  
2 title="Structured Query Language">SQL</abbr>
```

The most basic HTML elements are things like `<p>` (paragraph), `` (unordered list) and `<h1>` (largest page header), which correspond closely to the kinds of elements you'd use to structure a [book](#) or an [academic webpage](#). The reason for this is that HTML is derived from SGML, which was used to do hierarchical mark up on raw text for book formatting. While academic webpages map fairly well to this hierarchical metaphor (a nested directory of chapters, sections, and subsections), many web pages have significant two-dimensional structure. /You will not go wrong if you think of a webpage to first order² as a set of non-overlapping rectangular boxes called `<div>` elements ([example](#))./

Since the advent of [HTML5](#), the major vendors (Google, Apple, Mozilla, Opera, Microsoft) have equipped their browsers to work with new tags like `<nav>` (navigation links) and `<section>` (article sections). These tags provide shortcuts and semantic meaning. Search engines can do more with `<section>` than `<div>` or even `<div class='my-section'>`, as the former tag gives some semantic information (i.e. that it is a section of an article) while the `<div>` tag alone just specifies that it's a box on a webpage, which could be anything from a login form to a container for a profile photo. In addition to these semantic tags, modern browsers understand new tags like `<video>`, `<audio>`, and `<canvas>`, which go well beyond the book metaphor. Here's a simple HTML page with the major elements `DOCTYPE`, `<head>`, and `<body>`:

The `DOCTYPE` is an incantation that tells the browser what [version](#) of HTML is being used. The `<head>` contains metadata on the page, which in this case includes the `<title>` and the [character encoding](#) (in this case `utf-8`). And the `<body>` contains the meat of the page content, which in this case is just a simple header and paragraph. Finally, here is how it [renders](#) in the browser.

²This is only "to first order", because you can force `<div>` elements to overlap in the x/y plane by the use of [negative margins](#) or to layer on top of each other using the [z-index property](#).

```

1  <!DOCTYPE html>
2  <html>
3
4      <head>
5          <title>Hello</title>
6          <meta charset="utf-8">
7      </head>
8
9      <body>
10         <h1>Header</h1>
11         <p>Hello world!</p>
12     </body>
13
14 </html>

```

CSS: Look and Layout

As noted above, while academic webpages tend to use unstyled HTML and map well to the hierarchical book metaphor, most websites today employ some degree of two-dimensional layout, coloration, and typographical customizations. CSS allows you to take the same HTML skeleton and clothe it in different fonts, background colors, and layouts. An early demonstration of the power of CSS was [CSS Zen Garden](#), which showed in principle that you could achieve a clean separation between content and appearance. In practice, several of the designs on CSS Zen Garden are actually very layout sensitive, with images micro-optimized for a particular layout or text heading, but the principle is mostly intact.

Let's take our sample webpage and add some extremely basic CSS to a `<style>` tag in the `<head>` section. Below, we're setting the font size of text within the `<p>` element to 14px and the color to red ([see it live](#)).

```

1  <!DOCTYPE html>
2  <html>
3
4      <head>
5          <title>Hello</title>
6          <meta charset="utf-8">
7          <style type="text/css">
8              p {
9                  font-size: 14px;
10                 color: red;
11             }
12         </style>
13     </head>
14
15     <body>
16         <h1>Header</h1>
17         <p>Hello world!</p>

```

```
18   </body>
19
20  </html>
```

Note that `<h1>` appeared large and bold without specifying anything custom in the `<style>` element. This happens because browsers specify different default styles for each element; moreover these styles can be subtly different. People work around browser inconsistencies by using [CSS resets](#), that zero out all the default settings and normalize them to the same values across browsers. Here is what the same example [looks like](#) with the *Normalize CSS* option clicked on the left hand side.

In practice it's a lot of effort to build up a reasonably good looking CSS stylesheet, as it involves everything from the layout of the page to the dropshadows on individual buttons. Fortunately there are now excellent CSS frameworks like [Bootstrap](#), which provide built-in [grid systems](#) for 2D layout, typography, element styling, and more. Here's that [same example](#) again, except now without the Normalize CSS option and with Bootstrap's default CSS instead. Bootstrap is an extremely good way to get up and running with a new website without reinventing the wheel, and can be readily styled with various themes once you get some traffic.

JS: Dynamics and Behavior

Once you've prettified an HTML page with CSS, it's time to make it dance with JS. In the early days of the web, Javascript was used for popping up [alert boxes](#) and simple special effects. Today JS is used for things like:

1. Confirming that the text in an email field is a [valid email](#).
2. Creating various input widgets like [autocomplete search boxes](#).
3. Pulling content from remote servers to create [dynamic news feeds](#).
4. Playing [games](#) in your browser.
5. Embedding API functionality to provide things like [payments](#) or [social integration](#).

The reason JS rather than Python or Haskell is used for these purposes is that a JS interpreter is present in every web browser (Chrome, Firefox, IE, Safari, Opera), and all elements of a webpage can be manipulated with Javascript APIs. Here's an [example](#) with the video tag, and here's an example of [changing](#) the CSS style on an HTML tag with JS. Furthermore, in addition to being present in web browser ("client-side") environments, extremely good command line ("server-side") implementations of Javascript are now available, like node.js. JS is now also available in databases like MongoDB, and new libraries are constantly ported to JS. JS is, in other words, the "Next Big Language" as prophesied by Steve Yegge. If you only know one language, you should know Javascript in all its manifestations.

Separation of concerns

Any web app will include HTML, CSS, and JS files in addition to server-side code. One of the advantages of node.js is that the server-side code is also in JS.

In principle, you should scrupulously separate your HTML, CSS, and JS, such that you can change any of the three (or more) files that specify a webpage without worrying about the other two. In practice, you will need to develop all three files (.html, .css, and .js) at the same time as significant changes to the structure of a page will often remove `id` attributes or CSS styles that your JS code depends upon.

Sometimes you will have a choice as to whether you implement a particular function in HTML, CSS, or JS, as they are not entirely independent. For example, it's possible to implement buttons via [pure CSS](#) or [JS](#), or to put JS invocations [within](#) HTML tags rather than using [JQuery](#). When in doubt, err on the side of separation of concerns: you should try to use HTML for structure, CSS appearance, and JS for behavior.

Tools

As you edit HTML/CSS/JS, the single most important thing to learn are the [Chrome Developer Tools](#). You can pull these up by opening up any page and then going to View -> Developer -> Developer Tools (on a Mac, the keyboard shortcut is Command-Option-I). With these tools, you can do the following:

1. Edit the styles and nodes live on a page.
2. Click an element on a page to find it in the markup, or vice versa.
3. Watch network connections and inspect HTTP requests and responses.
4. Execute Javascript to alter the page.
5. Mimic mobile browsers with the [User Agent Switcher](#).

In addition, you should familiarize yourself with [jsfiddle](#) (very useful for sharing examples) and reloading tools like [Live Reload](#) or [Codekit \(more\)](#).

Deployment, DNS, and Custom Domains

Deployment: Dev, Staging, and Production

The easiest way to edit a website is to simply edit code on the live server for that website. That means SSHing directly into the server for `example.com`, loading up `emacs` or `vim`, editing the HTML/CSS/JS, and then just saving the file and reloading `example.com`. This style of web development is commonly practiced by graduate students on academic websites and does have the advantage of extreme simplicity. However, there are several problems with it. If there is only one copy of the codebase and it is located on the production website, it becomes difficult to:

- test features before end users see them
- roll back code that has introduced bugs
- restore code or data from catastrophic crashes of the server
- incorporate contributions from multiple engineers
- perform A/B testing of different incompatible features

We can solve these problems by introducing the concepts of (1) distributed version control of the codebase (e.g. via `git`) and (2) a dev/staging/production flow. The use of distributed version control allows us to save, restore, test, and merge multiple versions of a *codebase* by making use of `git branch` ([1](#), [2](#), [3](#), [4](#)). Combining this with separate servers for dev/staging/production allows us to thoroughly test changes to a *website* before rolling them out to end users. In a bit more detail, here's how our dev/staging/production flow (Figures [1](#) and [2](#)) will work in the class:

1. Development.

- You will develop on an SSH/HTTP/HTTPS-accessible EC2 machine with a text editor like `emacs`, and preview your work in the browser, using Chrome User Agent Switcher and Resolution Test to mimic mobile browsers.
- The `screen` utility will be used to preserve your remote session so that you can disconnect from your dev instance and then reconnect from the same or another machine.
- You will save your website's codebase along the way in a `git` repository (say `example-site.git`), hosted on Github. Importantly, and this is a new concept, you will use `git branch` to create a `develop` branch of your codebase rather than making edits directly to the `master` branch.

- You will also save the commands required to set up the EC2 machine (`setup.git`) and configure your development environment (`dotfiles.git`) in git repositories, to make the entire development process highly reproducible. This will permit other engineers to run/edit/debug your code and get exactly the same results.
- As you make changes in `develop`, you will create/commit/push edits, and run a local web server to permit the previewing of these edits in a browser.

2. *Staging.*

- Once you have a “release candidate”, that is, a version of the code that you think is worthy of evaluating for the production website, you will merge the code into a `staging` branch in `example-site.git` and optionally use `git tag` to tag a release candidate, with a systematic name like `staging-june-13-2013-r01`.
- Using Heroku’s [syntax for setting up staging environments](#), you will then push to Heroku. This gives us a clone of exactly how the live site will look, as it might be subtly different from the EC2 dev preview.
- In our class, `staging` is a separate Heroku app available at something like <http://example-site-staging.herokuapp.com>.
- We may find some last minute bugs; if so, we make these edits/commits in the `develop` branch and then merge them into `staging`, rather than editing `staging` directly. This forces discipline; in particular it ensures that your bug fix is not incompatible with changes made to `develop` by others since your deploy to `staging`.

3. *Production.*

- Once you and your automated and manual testers confirm that the site works as intended on the staging server (e.g. by clicking and trying to break things, or running headless tests against the staging URL), then you have a confirmed release candidate of `example-site.git` in the `staging` branch. This is now merged into `master`.
- You may wish to further distinguish this revision by using another `git tag` with an internal naming scheme (e.g. `deploy-june-13-2013`).
- You will then push this revision to the production server, which is viewed by end users. For the purposes of this class, that is a Heroku app like <http://example-site.herokuapp.com>.
- Finally, if you have set up a custom domain, this is when end users see your changes propagate to `example.com` (see the last section in this lecture).

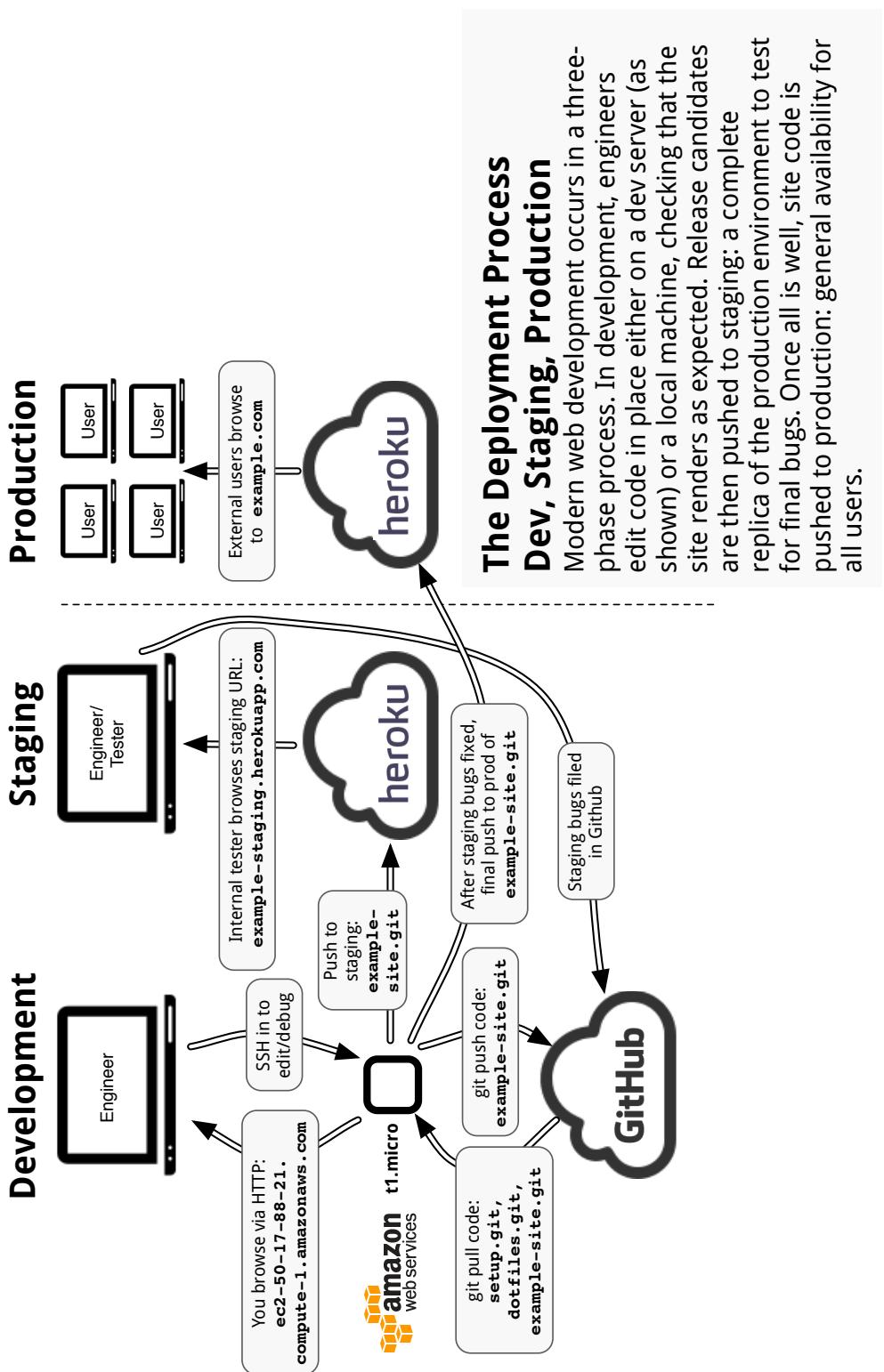


Figure 1: Modern web development proceeds in three phases: dev, staging, and production.

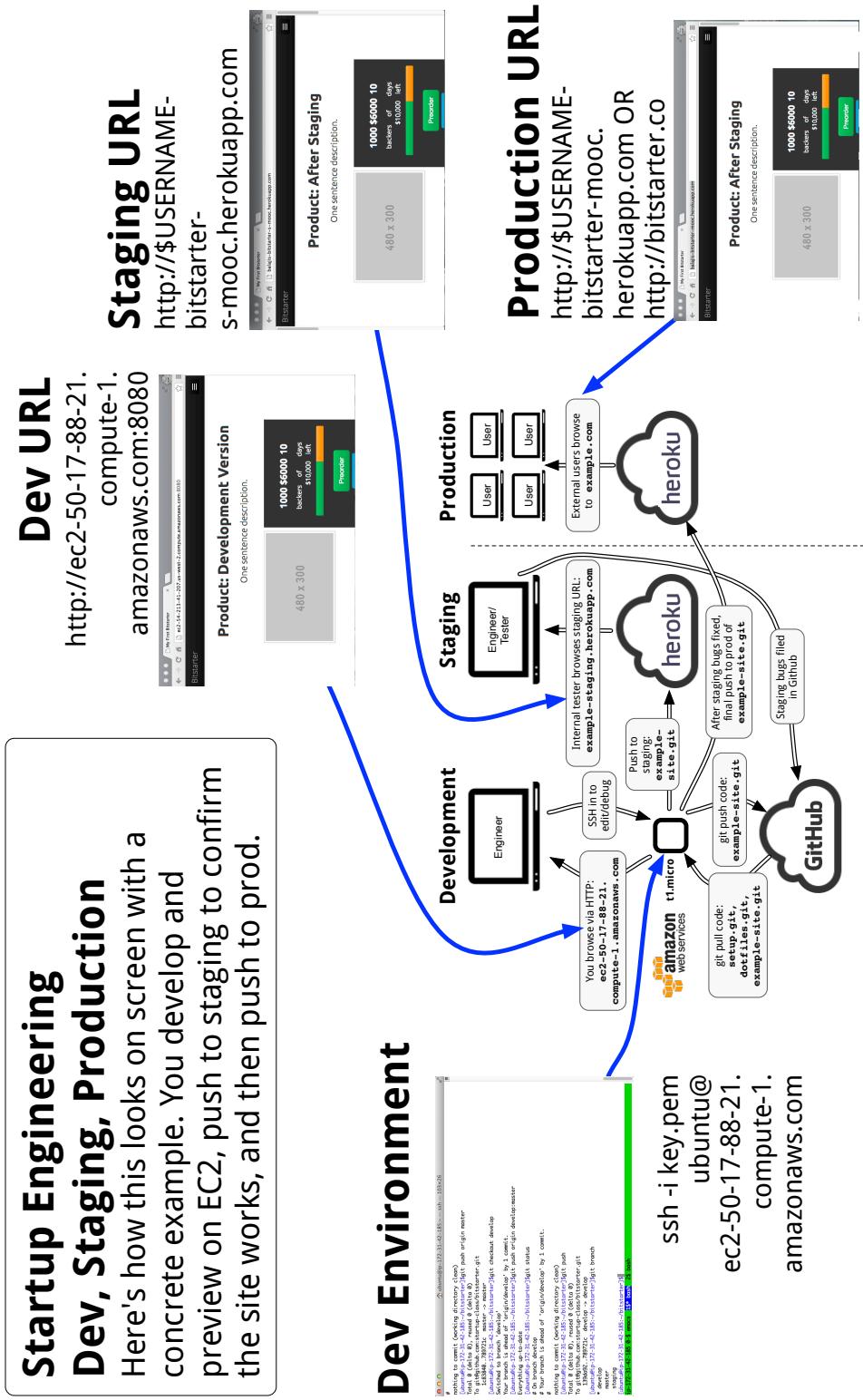


Figure 2: Here's how dev/staging/prod looks on screen. See also Figure 1.

Sidebar: Comparing EC2 vs. local laptops for development

Note that for the class we use EC2 as both editing environment and dev server, and Heroku as our staging and production servers. We do this for several reasons:

1. *Standardized environment.* As noted in Lecture 2, EC2 will work for any student with a credit card and evens out the background differences between student machines. In subsequent versions of the class we may also support local VMs, which are similar in concept.
2. *Shareable development URLs.* EC2 instances make it easy to set up a development server on the public internet (e.g. at `http://ec2-54-213-41-207.us-west-2.compute.amazonaws.com:8080` or the equivalent). This is very convenient for testing out certain kinds of apps (especially mobile apps that need a dev API) or sharing in-development URLs with people outside your organization. It's possible to do something like this with a laptop but harder.
3. *Scriptable infrastructure.* Terminating and restoring pristine EC2 instances also introduces the crucial concept of scriptable infrastructure; think of our simple `setup.sh` as the beginning of `devops`. Your dev environment needs to be scripted for it to be reproducible by other engineers, and you're not going to wipe/restore someone's local laptop every time just to ensure that (e.g.) you have exactly the same PATH settings, library versions, and the like. You can run a VM locally, but that is somewhat slow. Thus EC2 provides development discipline.
4. *Portable, general environment.* Learning how to work in a Linux terminal development environment means you can be productive in a new language or toolchain fairly rapidly. The differences between different emacs or vim modes are trivial relative to the differences between dedicated C# or Python editors. This is important as many projects require the use of multiple languages or coding across multiple environments.
5. *Large or private datasets.* When working with large datasets or databases with privacy constraints, sometimes you can download a small and/or anonymized subsample that is suitable for offline debugging. However, if that proves infeasible (as it is for search, social, genomics, advertising, or many other big data problems), you need to be able to work on a remote server (either one in your own colo or datacenter, or on EC2 or a similar IAAS provider). This experience will be very similar to the class experience, where we are just using a local laptop as a thin client.

That said, in practice (outside the class) engineers often choose to develop on a local laptop for convenience when possible (Figure 3), while previewing locally by connecting on `localhost`. At the risk of belaboring the obvious: if you can get things to work on your local machine and don't want to use EC2 for development, then go ahead and do that. We are simply presenting one solution that works; you should feel free to invent your own tech stack, use the tools you're already familiar with, or mix and match some of our choices with some of yours. However, given the size of the class we will not be able to give technical support for nonstandard configurations; we hope you understand!

Alternative: Develop Locally

You can use a local machine, and preview locally. Advantage: offline use. Disadvantage: lack of scriptable dev environment, can't easily handle large datasets, less similar to production.

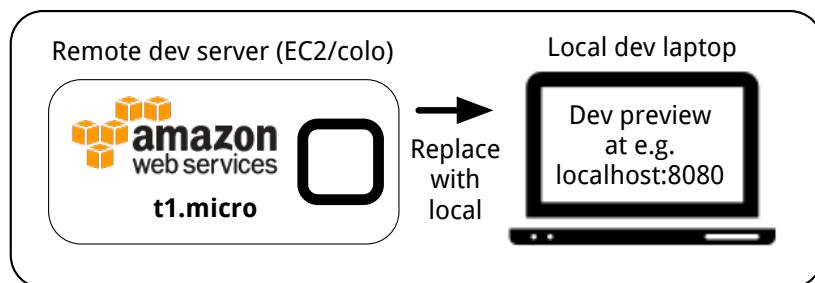


Figure 3: You can replace the remote dev instance with a local laptop. This has advantages and disadvantages; see main text.

Preliminaries: SSH/HTTP/HTTPS-accessible EC2 dev instance

We're now going to do a worked example of dev/staging/production with your code from HW3. To allow local previewing of your files under development, we'll first set up a dev instance which allows HTTP connections not just on port 22 (which handles SSH traffic), but also port 80 (which handles HTTP), port 443 (which handles HTTPS) and port 8080 (a typical debug/dev port for HTTP). This will allow you to edit code on EC2 and debug it by previewing in a browser, rather than copying it locally or always pushing to Heroku. To set this up you will want to go to the AWS Dashboard, set up a new security group, and then launch an instance¹ with that security group. Take a look at Figures 4-14.

The screenshot shows the AWS EC2 dashboard with the 'Security Groups' section selected. A red box highlights the 'Create Security Group' button at the top of the main content area. Below it is a table listing two existing security groups:

	Group ID	Name	VPC ID	Description
<input type="checkbox"/>	sg-b15eb9de	default	vpc-c7fe63af	default VPC security group
<input type="checkbox"/>	sg-1aa64075	quicklaunch-1	vpc-c7fe63af	quicklaunch-1

Figure 4: Go to the EC2 dashboard and create a new security group.

¹You can also change the security group of an existing instance without starting up a new one, but it's pedagogically easier to just show how to do this from scratch rather than resuming (as everyone's VM will be in a different state).

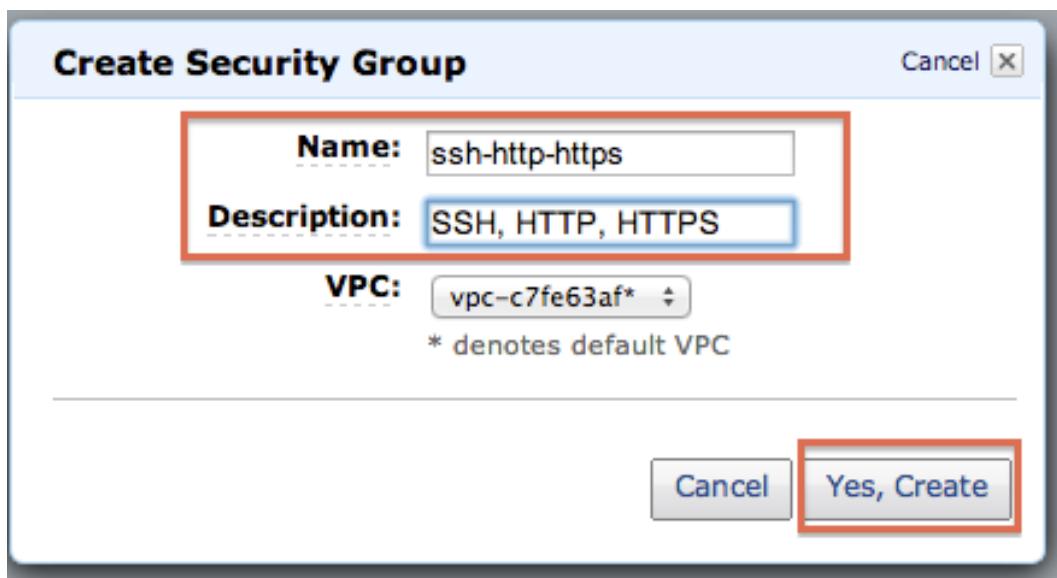


Figure 5: Call it ssh-http-https.

	Group ID	Name	VPC ID	Description
<input type="checkbox"/>	sg-b15eb9de	default	vpc-c7fe63af	default VPC security group
<input type="checkbox"/>	sg-1aa64075	quicklaunch-1	vpc-c7fe63af	quicklaunch-1
<input checked="" type="checkbox"/>	sg-697e9f06	ssh-https	vpc-c7fe63af	SSH, HTTP, HTTPS

1 Security Group selected

Security Group: ssh-https

Details **Inbound** Outbound

Group Name: ssh-https
 Group ID: sg-697e9f06
 Group Description: SSH, HTTP, HTTPS
 VPC ID: vpc-c7fe63af

Figure 6: Click it and click the 'Inbound' tab.

TCP Port (Service)	Source	Action
22 (SSH)	0.0.0.0/0	Delete

Figure 7: Click the dropdowns in sequence and open up the HTTP and HTTPS ports.

1 Security Group selected

Security Group: ssh-ssh-~~http~~-https

Inbound

Create a new rule: Custom TCP rule

Port range: 8080 (e.g., 80 or 49152-65535)

Source: 0.0.0.0/0 (e.g., 192.168.2.0/24, sg-47ad482e, or 1234567890/default)

**SSH, HTTP, HTTPS are built in options
But for port 8080 you need a
Custom TCP rule as shown.**

Add Rule

Apply Rule Changes

Figure 8: For port 8080, you will need to set up a custom TCP rule, as shown.

1 Security Group selected

Security Group: ssh-ssh-https

Inbound*

TCP Port (Service)	Source	Action
22 (SSH)	0.0.0.0/0	Delete
80 (HTTP)	0.0.0.0/0	Delete
443 (HTTPS)	0.0.0.0/0	Delete
8080 (HTTP*)	0.0.0.0/0	Delete

Your changes have not been applied yet.

Apply Rule Changes

1) This is what your final list of rules should look like.
2) Then click here.

Figure 9: You should have four ports open at the end: 22, 80, 443, and 8080.

https://console.aws.amazon.com/ec2/home?region=us-west-2#s=Instances

Services ▾ Edit ▾ Startup Stanford ▾ Oregon

EC2 Dashboard
Events
Tags

INSTANCES Instances

Launch Instance Actions ▾

Viewing: All Instances All Instance Types Search

1 to 1 of 1 Instances

Name	Instance	AMI ID	Root Device	Type	State	Status
empty	i-d82525ed	ami-70f96e40	ebs	t1.micro	running	✓ 2

AMIs
Bundle Tasks

ELASTIC BLOCK STORE Volumes
S snapshots

NETWORK & SECURITY Security Groups
Elastic IPs
Placement Groups
Load Balancers
Key Pairs
Network Interfaces

Figure 10: Now terminate your old instances and launch a new instance (it is also possible to apply the security group to a running instance).

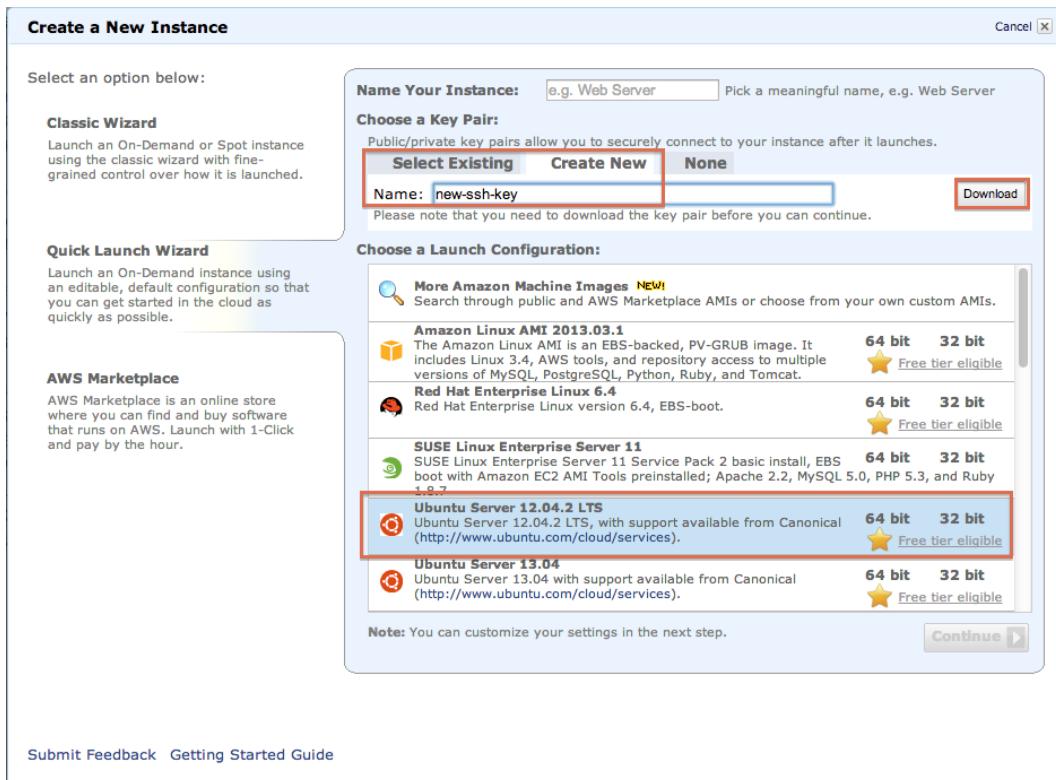


Figure 11: Pick Ubuntu 12.04.2 LTS and download the pem key as usual.

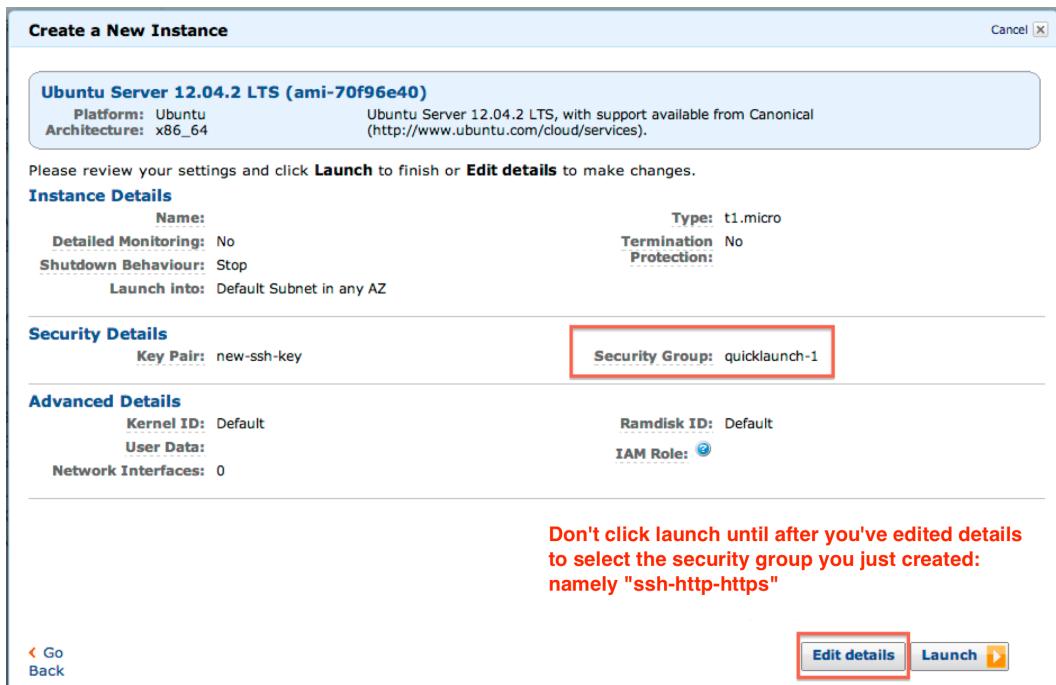


Figure 12: Make sure to click 'Edit details' as the security group you just configured probably will not be selected by default.

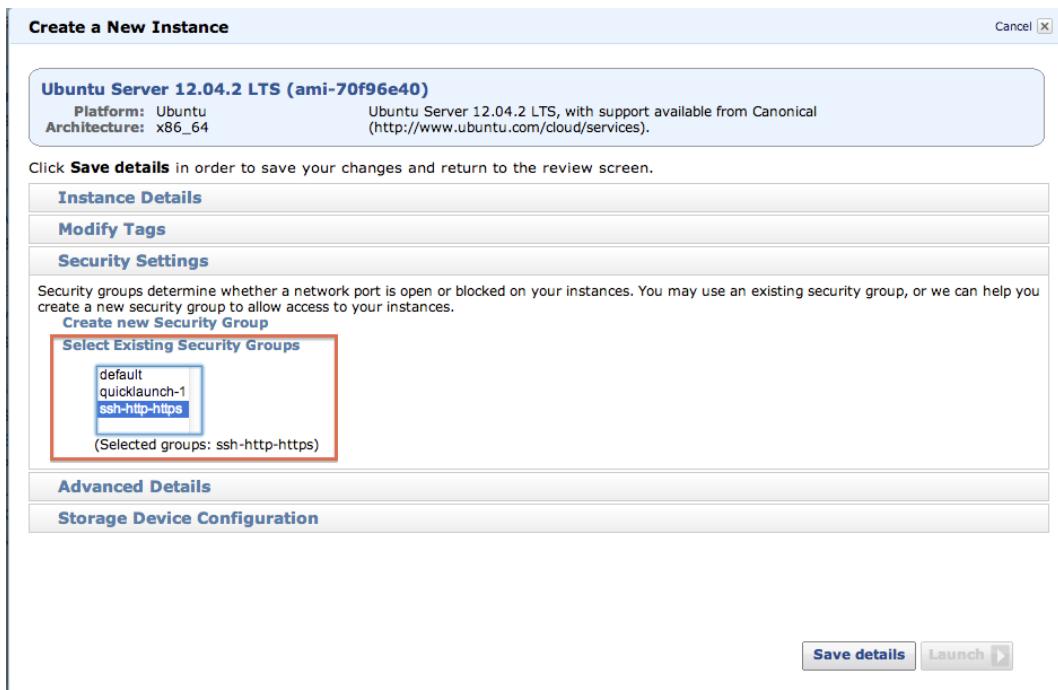


Figure 13: Pick *ssh-http-https* in the Security Settings accordion.

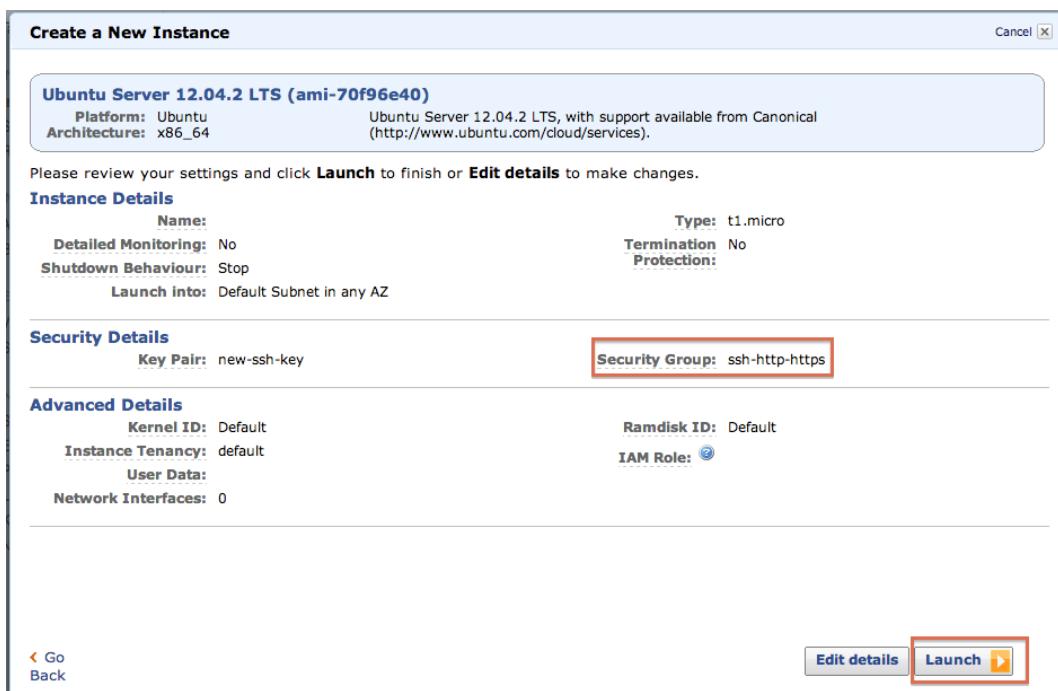


Figure 14: Now launch your instance.

You'll now want to run `setup.sh` to reinstall everything (including `dotfiles.git`) and then use `git clone` to pull down your bitstarter repository that you created in from github. Now edit your `bitstarter/web.js` file to configure it to port 8080 and then run `node web.js`. You should now be able to see your `index.html` in the browser, as shown in the following screenshots:

The screenshot shows a terminal window titled "ubuntu@ip-172-31-36-95:~/bitstarter — ssh — 80x24". The user has run `cat web.js` to view the current code. The code defines an Express app that serves the contents of `index.html` on port 8080. A red box highlights the line `var port = process.env.PORT || 8080;` with the annotation "Make this edit and save." A second red box highlights the output of the command `node web.js`, which shows the application is listening on port 8080.

```
ubuntu@ip-172-31-36-95:~/bitstarter]$ cat web.js
var express = require('express');
var fs = require('fs');
var htmlfile = "index.html";

var app = express.createServer(express.logger());

app.get('/', function(request, response) {
  var html = fs.readFileSync(htmlfile).toString();
  response.send(html);
});

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log("Listening on " + port);
});
[ubuntu@ip-172-31-36-95:~/bitstarter]$ node web.js
Listening on 8080
```

Figure 15: Change your `web.js` to serve on port 8080, and run `node web.js`

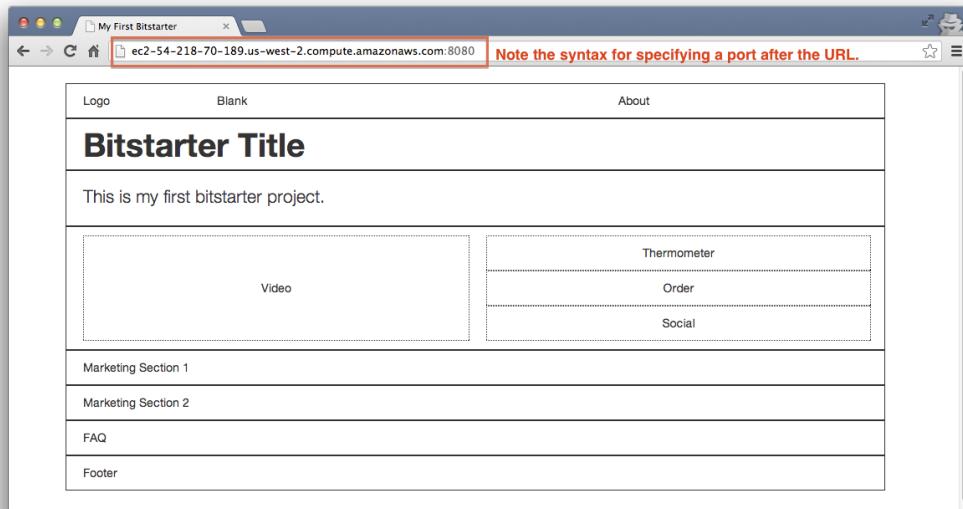


Figure 16: Confirm that you can view your site in a browser.

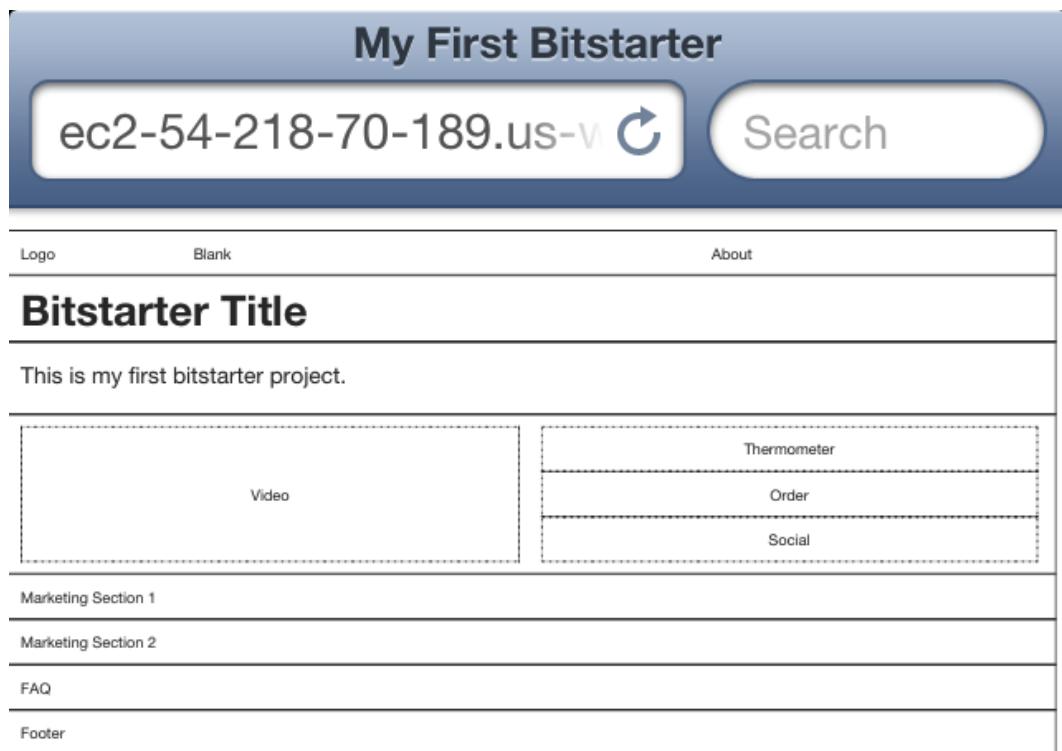


Figure 17: Confirm that you can view your site in a mobile browser.

Note that the last screenshot shows a mobile phone browsing to the URL. This works because by opening up the security group, you made your EC2 instance accessible over port 8080 over the public internet. However, it's a little inconvenient to keep refreshing your mobile phone. Instead, one of the best ways to preview how your website will look with a mobile client is by installing Chrome's [User Agent Switcher](#) and its [Resolution Test](#) tools. Then navigate to the same URL and set things up as shown in Figures 18-19. One way to confirm that you've configured this properly is to go to [airbnb.com](#) in Chrome with the User Agent Switcher set to iPhone 4 and see if you get automatically redirected to [m.airbnb.com](#).

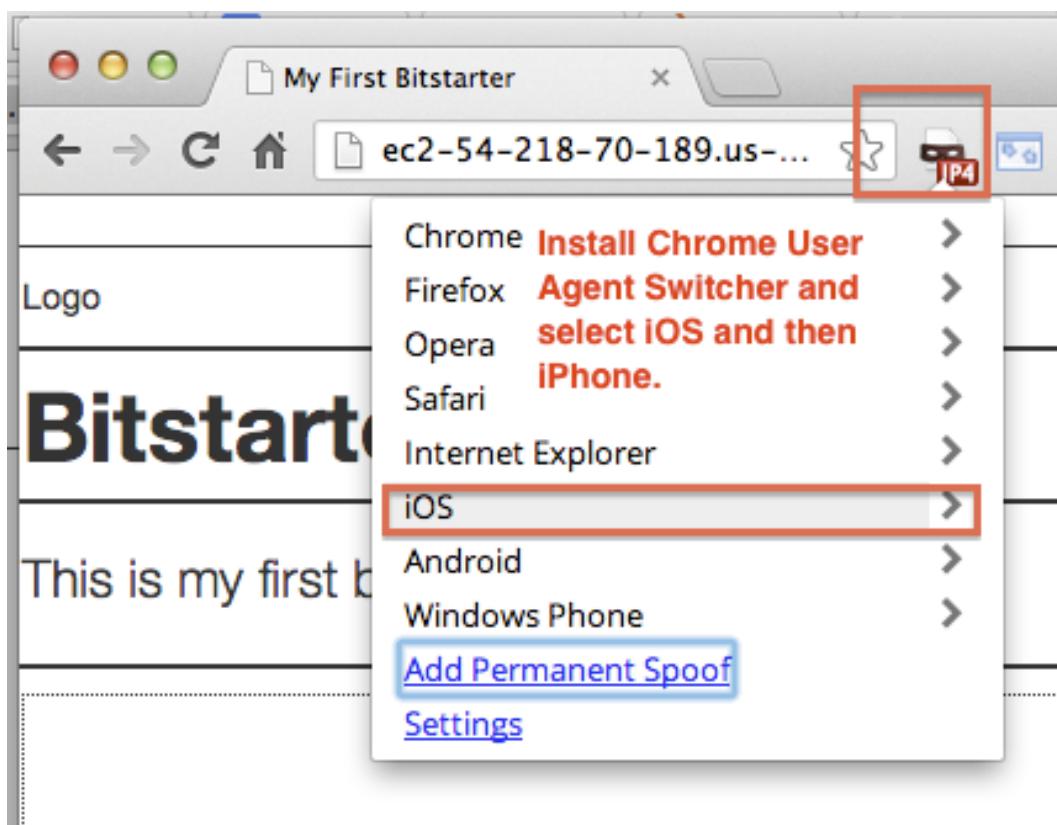


Figure 18: To do mobile site previews on your desktop, set up Chrome's User Agent Switcher and select iOS -> iPhone 4.

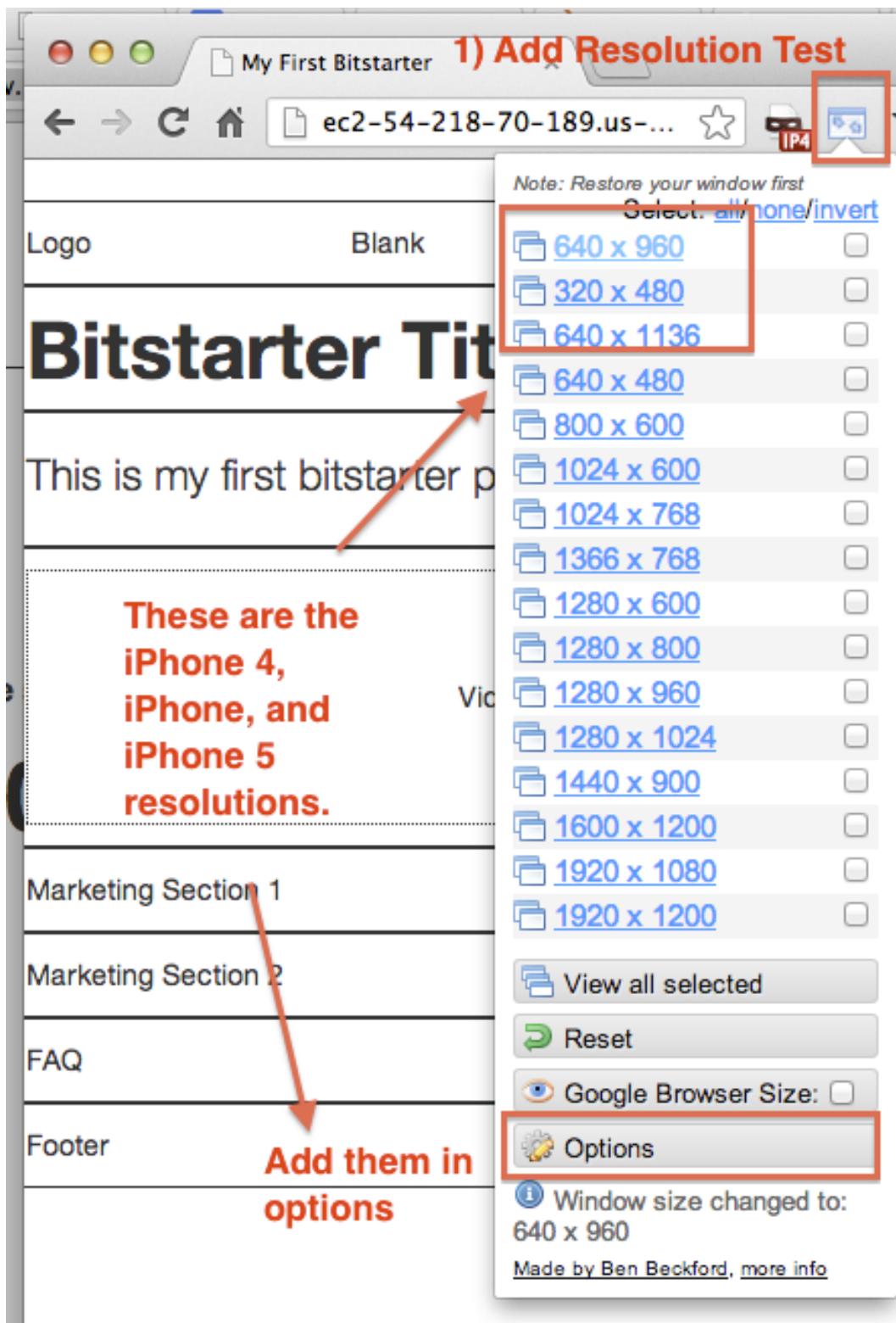


Figure 19: To get the screen to the right resolution, use the Resolution Test Chrome extension as shown.

At this point you've just set up a dev instance capable of editing websites and previewing both desktop and mobile versions over port 8080.

Creating and managing git branches

One of the most useful features of git is that it allows you to manage many simultaneous edits to multiple versions of a codebase. Suppose you have a production website and you want to add features. With git, a single engineer can use a three branch model (Figure 20) to make sure that all new features are thoroughly tested in dev and staging before being pushed to production. Moreover, this model [scales up](#) to support many engineers independently making non-overlapping changes to that website, previewing them on separate staging servers, and then interleaving them one-by-one, all while the production website remains unchanged. The fundamental tool for this process is the `git branch` command. Please read the following [reference](#) and [1](#), [2](#), [3](#), [4](#) as supplements; this will help you follow along with the screenshots in the next section.

Git Branching Model

Dev, Staging, Production

For the purposes of the class, we'll use a three branch model to implement the dev/staging/production flow. You will do primary edits of code in the **develop** branch. When you have a release candidate, git merge these edits into the **staging** branch and push to the staging server to preview what the live site should look like. Finally, when that looks good, we git merge the changes into the **master** branch and push to the production server.

The overall concept is that edits should flow one way: from the **develop** to the **staging** and then to the **master** branch. Bugs get caught before they hit the live site, and you can now roll back to an earlier production release.

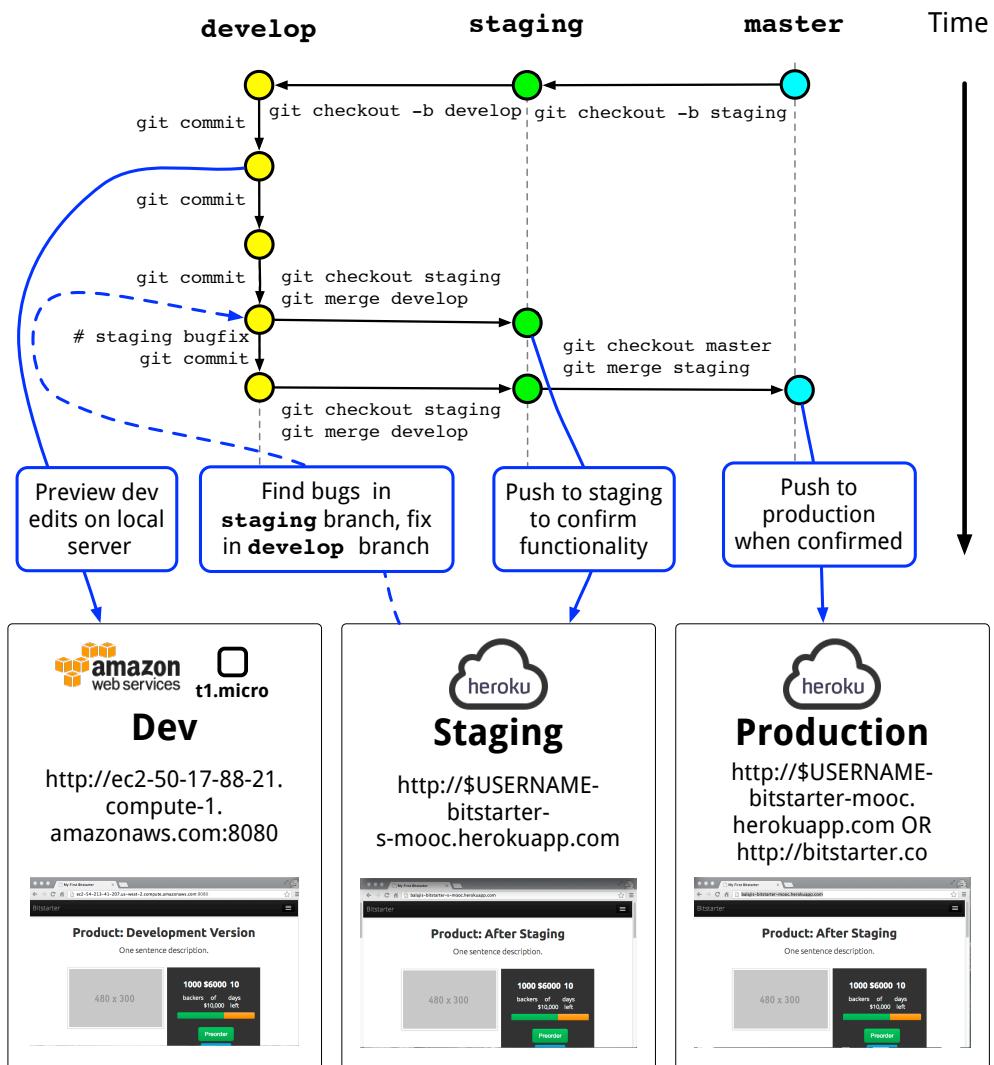


Figure 20: We'll be using the following relatively simple model to manage changes to the live site. See nvie.com for a more complex one.

Worked Example: Dev, Staging, and Production

Now that we've got our dev instance and understand a bit about git branching, we can illustrate the dev/staging/production flow. See the annotated interactive session and then the following screenshots (Figures 21-30). A few notes:

1. *Use your bitstarter repo.* Note that we have intentionally kept `github.com/startup-class/bitstarter` private for now to give you an incentive to do the interactive session yourself.
2. *Consult previous lectures.* Note also that the first portion of the interactive session below is straightforward given the previous lectures (especially Interactive Start and the Linux Command Line). The parts we are screenshotting begin at the `git checkout -b develop` line.
3. *Naming conventions.* When following along with the interactive session, please use the following naming convention for your Heroku apps. This will make it easy for us to manage on the Heroku end.

```
1 # Heroku apps have a thirty character limit
2 http://$YOUR_GITHUB_USERNAME-bitstarter-s-mooc.herokuapp.com # staging
3 http://$YOUR_GITHUB_USERNAME-bitstarter-mooc.herokuapp.com # production
```

Now let's go through the session. You should follow along, type in commands yourself, and review the screenshots (Figures 21-30) along the way.

```
1 # Assuming you launched a new EC2 instance with the ssh-http-https security
2 # group, run these commands to set up the machine.
3 sudo apt-get install -y git-core
4 git clone https://github.com/startup-class/setup.git
5 ./setup/setup.sh
6
7 # Next, create an SSH key and (by copy/pasting with the mouse)
8 # add it to Github at https://github.com/settings/ssh
9 ssh-keygen -t rsa
10 cat ~/.ssh/id_rsa.pub
11
12 # Now you can clone via SSH from github.
13 # Cloning over SSH allows you to push/pull changes.
14 # Note that you should substitute your own username and email.
15 git clone git@github.com:$USERNAME/bitstarter.git
16 git config --global user.name $USERNAME
17 git config --global user.email $EMAIL
18 exit # log out and log back in to enable node
19
20 # Next, change into the bitstarter directory and
21 # get all npm dependencies. Replace port 5000
22 # with 8080 if this is not already done, to allow
23 # serving over that port.
```

```

24 cd bitstarter
25 npm install
26 sed -i 's/5000/8080/g' web.js
27
28 # Create a development branch and push it to github
29 # The -b flag creates a new branch (http://git-scm.com/book/ch3-2.html)
30 # The -u sets that branch to track remote changes (http://goo.gl/sQ60I)
31 git checkout -b develop
32 git branch
33 git push -u origin develop
34
35 # Create a staging branch and push it to github
36 git checkout -b staging
37 git branch
38 git push -u origin staging
39
40 # Login and add the SSH key you created previously to Heroku
41 # Then create heroku apps.
42 #
43 # IMPORTANT: Heroku has a 30 character limit, so use the naming convention:
44 # GITHUB_USERNAME-bitstarter-s-mooc for your staging app
45 # GITHUB_USERNAME-bitstarter-mooc for your production app
46 #
47 # Also see:
48 # https://devcenter.heroku.com/articles/keys
49 # http://devcenter.heroku.com/articles/multiple-environments
50 heroku login
51 heroku keys:add
52 heroku apps:create $USERNAME-bitstarter-s-mooc --remote staging-heroku
53 heroku apps:create $USERNAME-bitstarter-mooc --remote production-heroku
54
55 # Now return to dev branch, make some edits, push to github.
56 #
57 # NOTE: we use 'git checkout develop' to simply change into the develop
58 # branch without creating it anew with the '-b' flag.
59 #
60 # NOTE ALSO: We use 'git push origin develop' rather than 'git push -u
61 # origin develop' because we already set up the branch to track remote
62 # changes once. This means that if others edit the 'develop' branch we can
63 # merge their changes locally with git pull --rebase.
64 git checkout develop
65 git branch
66 emacs -nw index.html
67 git commit -a -m "Edited product desc in index.html"
68 git push origin develop
69
70 # Start node server in another screen tab to preview dev edits in browser.

```

```

71 # We use /usr/bin/env to avoid any issues from our rlwrap alias. Note
72 # that you can also run this as a background process with
73 # /usr/bin/env node web.js &, though you will want to redirect
74 # the log output.
75 #
76 # Then, in your browser go to the equivalent of:
77 # http://ec2-54-213-41-207.us-west-2.compute.amazonaws.com:8080
78 # Also see the branch you just pushed at:
79 # https://github.com/USERNAME/bitstarter/tree/develop
80 /usr/bin/env node web.js
81
82 # Once you have made enough edits that you like, it's time to merge commits
83 # into staging and then push to the staging server.
84 # See here to understand the 'git push staging-heroku staging:master' syntax:
85 # http://devcenter.heroku.com/articles/multiple-environments
86 # http://git-scm.com/book/ch9-5.html
87 git branch
88 git checkout staging # not checkout -b, because branch already exists
89 git merge develop # merge changes from develop into staging.
90 git push staging-heroku staging:master
91
92 # Now inspect staging at USERNAME-bitstarter-s-mooc.herokuapp.com
93 # If you need to make edits, do them in develop, preview, and then merge into staging.
94 # Try to keep the flow of edits one way.
95 #
96 # Here is how we fix bugs detected in staging. First, we 'git checkout
97 # develop' again. We make fixes (with emacs), save (with git commit -a -m),
98 # push them to github (git push origin develop), and then return to staging
99 # (git checkout staging).
100 #
101 # Now that we are back in the 'staging' branch, a status we can check with
102 # 'git branch', we again issue the all-important 'git merge develop'. This
103 # merges changes from the develop branch into staging.
104 git checkout develop
105 emacs -nw index.html # make edits and save
106 git commit -a -m "Fixed staging bugs in develop branch."
107 git push origin develop # push develop commits to github
108 git checkout staging
109 git merge develop # merge changes from develop into staging.
110 git push origin staging # push staging commits to github
111 git push staging-heroku staging:master # push again to Heroku
112
113 # Once we confirm the website version deployed on staging works as intended,
114 # we merge into master and then push to production.
115 git checkout master
116 git merge staging
117 git push production-heroku master:master

```

```
118
119 # Assuming all goes well, go back to develop and edit some more.
120 git checkout develop
121 emacs -nw index.html
```

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout -b develop
Switched to a new branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push -u origin develop
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
 * [new branch]      develop -> develop
Branch develop set up to track remote branch develop from origin.
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout -b staging
Switched to a new branch 'staging'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
  develop
  master
* staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push -u origin staging
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
 * [new branch]      staging -> staging
Branch staging set up to track remote branch staging from origin.
```

Figure 21: Begin by moving into your bitstarter directory and creating a `develop` and `staging` branch. Push these to the `origin` as shown, which is Github in this case (see the `bitstarter/.git/config` file for details).

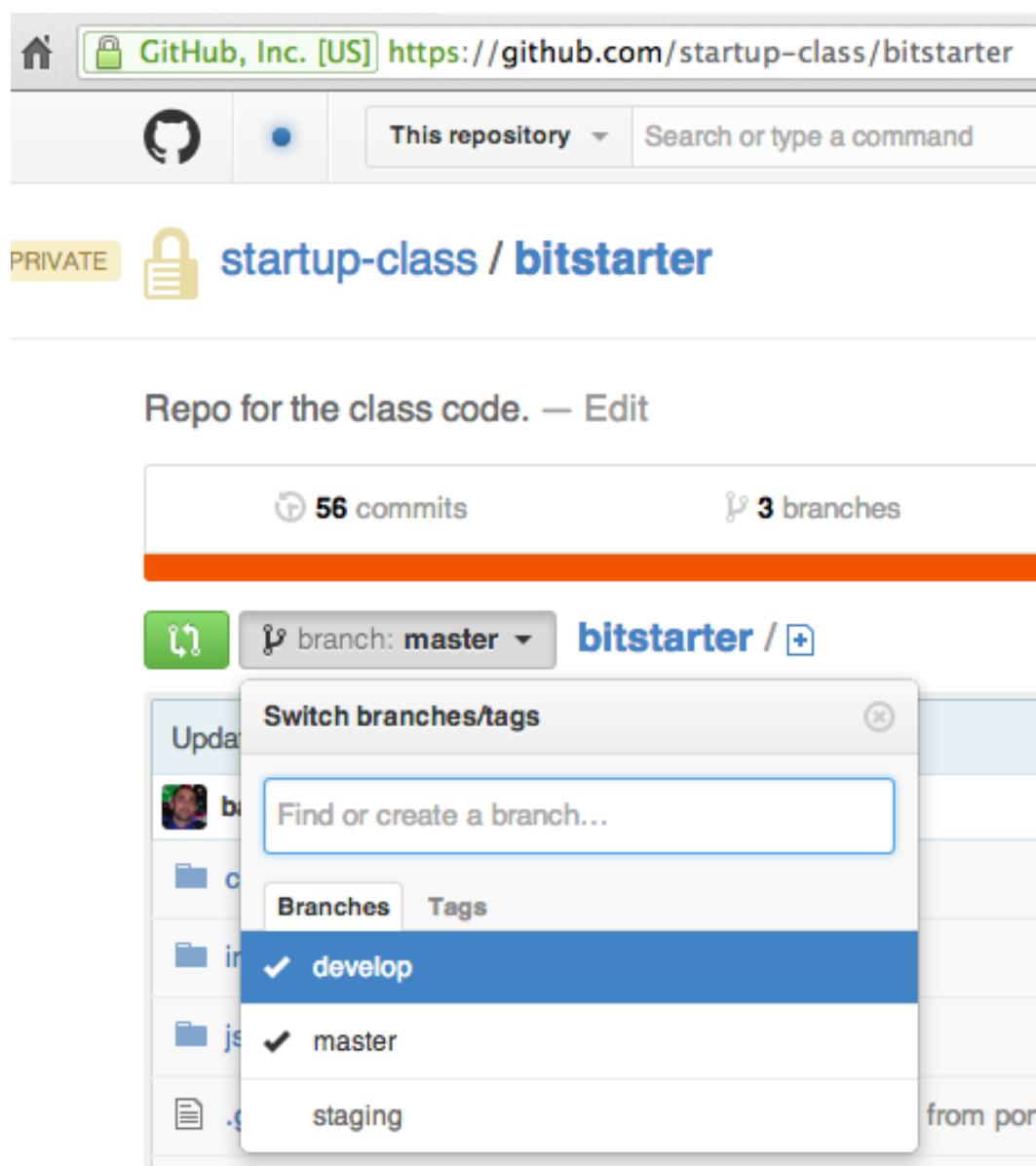


Figure 22: After you have created these branches, go to your github repo online at `github.com/USERNAME/bitstarter` and you should be able to see the branches, as shown.

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku login
Enter your Heroku credentials.
Email: balajis@stanford.edu
Password (typing will be hidden):
Authentication successful.
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku keys:add
Found existing public key: /home/ubuntu/.ssh/id_rsa.pub
Uploading SSH public key /home/ubuntu/.ssh/id_rsa.pub... done
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku apps:create balajis-bitstarter-s-mooc --remote staging-heroku
Creating balajis-bitstarter-s-mooc... done, region is us
http://balajis-bitstarter-s-mooc.herokuapp.com/ | git@heroku.com:balajis-bitstarter-s-mooc.git
Git remote staging-heroku added
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku apps:create balajis-bitstarter-mooc --remote production-heroku
Creating balajis-bitstarter-mooc... done, region is us
http://balajis-bitstarter-mooc.herokuapp.com/ | git@heroku.com:balajis-bitstarter-mooc.git
Git remote production-heroku added
```

Figure 23: Now return to the command line, log into heroku and add your SSH key. This is the same key you generated and pasted into Github (see previous commands in the interactive session). Next, create a staging and production app as shown.

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout develop
Switched to branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$sed -i 's/Product/Foo/g' index.html
[ubuntu@ip-172-31-34-196:~/bitstarter]$git diff
diff --git a/index.html b/index.html
index bcd823a..81ccc0e 100644
--- a/index.html
+++ b/index.html
@@ -133,7 +133,7 @@
    <!-- www.google.com/fonts/specimen/Ubuntu#pairings -->
    <div class="row-fluid heading">
        <div class="span12">
-           <h1>Product: Development Version</h1>
+           <h1>Foo: Development Version</h1>
        </div>
    </div>
    <div class="row-fluid subheading">
```

[ubuntu@ip-172-31-34-196:~/bitstarter]\$git commit -a -m "Updated product description."
[develop d20a72f] Updated product description.
1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]\$git push origin develop
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 300 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
4257b98..d20a72f develop -> develop_

Figure 24: Let's now simulate the development process. We make an edit to a file, commit that edit, and then push these changes to Github.

A screenshot of a terminal window titled "ubuntu@ip-172-31-34-196:~/bitstarter — ssh — 70x12". The window contains the following text:

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$ /usr/bin/env node web.js
Listening on 8080
```

The bottom status bar shows the IP address "ip-172-31-34-196" and two bash sessions: "0\$ emacs" and "1\$* bash".

Figure 25: Start up your node server in one screen tab...

A screenshot of a terminal window titled "ubuntu@ip-172-31-34-196:~/bitstarter — ssh — 70x12". The window contains the following text:

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$ git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$ git status
# On branch develop
nothing to commit (working directory clean)
[ubuntu@ip-172-31-34-196:~/bitstarter]$
```

The bottom status bar shows the IP address "ip-172-31-34-196" and two bash sessions: "0\$ emacs" and "1\$* bash".

Figure 26: ... and continue editing in another.

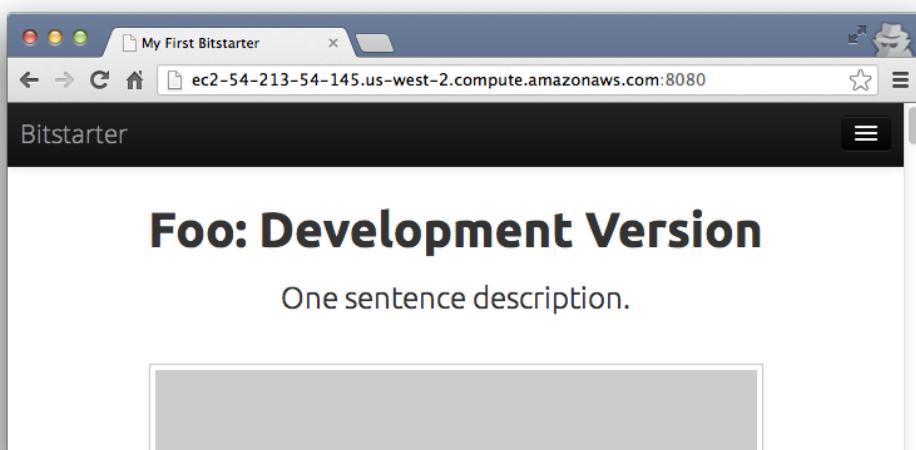


Figure 27: Now you can preview your *develop* branch edits in the browser.

```

[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout staging
Switched to branch 'staging'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git merge develop
Updating 4257b98..d20a72f
Fast-forward
 index.html |    2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push staging-heroku staging:master
The authenticity of host 'heroku.com (50.19.85.156)' can't be established.
RSA key fingerprint is 8b:48:5e:67:0e:c9:16:47:32:f2:87:0c:1f:c8:60:ad.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'heroku.com,50.19.85.156' (RSA) to the list of known hosts.
Counting objects: 199, done.
Compressing objects: 100% (90/90), done.
Writing objects: 100% (199/199), 265.19 KiB, done.
Total 199 (delta 107), reused 194 (delta 105)

-----> Node.js app detected
-----> Resolving engine versions

[snipped]

-----> Compiled slug size: 4.2MB
-----> Launching... done, v4
      http://balajis-bitstarter-s-mooc.herokuapp.com deployed to Heroku
      To git@heroku.com:balajis-bitstarter-s-mooc.git
      * [new branch]      staging -> master
[ubuntu@ip-172-31-34-196:~/bitstarter]$

```



Figure 28: When you feel you have a release candidate, you `git checkout` the `staging` branch and merge in the commits from `develop`, as shown. Then you push this `staging` branch to the `master` branch of the `staging-heroku` remote, which we set up with the `heroku create` command earlier. The syntax here is admittedly weird; see [here](#) and [here](#) for details.

```

[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout develop
Switched to branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$sed -i 's/Foo:/Bar:/g' index.html
[ubuntu@ip-172-31-34-196:~/bitstarter]$git diff
diff --git a/index.html b/index.html
index 81ccc0e..9ef6f5d 100644
--- a/index.html
+++ b/index.html
@@ -133,7 +133,7 @@
<!-- www.google.com/fonts/specimen/Ubuntu#pairings -->
<div class="row-fluid heading">
<div class="span12">
-      <h1>Foo: Development Version</h1>
+      <h1>Bar: Development Version</h1>
    </div>
  </div>
<div class="row-fluid subheading">
[ubuntu@ip-172-31-34-196:~/bitstarter]$git commit -a -m "Fixed bug found in staging. Committing to develop branch."
[develop eddbb40] Fixed bug found in staging. Committing to develop branch.
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push origin develop # push to github
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 324 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
  d20a72f..eddbb40 develop -> develop
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout staging
Switched to branch 'staging'
Your branch is ahead of 'origin/staging' by 1 commit.
[ubuntu@ip-172-31-34-196:~/bitstarter]$git merge develop
Updating d20a72f..eddbb40
Fast-forward
 index.html | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push origin staging
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
  4257b98..eddbb40 staging -> staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push staging-staging-heroku staging:master

```

[snipped]

```

----> Compiled slug size: 4.2MB
----> Launching... done, v5
      http://balajis-bitstarter-s-mooc.herokuapp.com deployed to Heroku
To git@heroku.com:balajis-bitstarter-s-mooc.git
  d20a72f..eddbb40  staging -> master

```



Figure 29: Now let's say upon viewing `staging` we determine there is a bug - the text `Foo` should have been `Bar`. We go back to `develop`, make an edit and then push these commits to github with `git push origin develop` as before. When we are done fixing the bug, we again merge `develop` into `staging` and push to `staging-heroku`. And now we can see that the bug is fixed at the `staging` URL. The important thing here is that we are using `staging` as a “dumb” branch which only merges in commits from `develop` rather than incorporating any edits of its own. This is generally good practice: commits should flow from `develop` to `staging` to `master` in a one-way flow (though you can make this model more sophisticated to account for e.g. `hotfixes` and the like).

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
  develop
  master
* staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout master
Switched to branch 'master'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git merge staging
Updating 4257b98..eddbb40
Fast-forward
 index.html | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push production-heroku master:master
```

[snipped]

```
-----> Launching... done, v4
http://balajis-bitstarter-mooc.herokuapp.com deployed to Heroku
To git@heroku.com:balajis-bitstarter-mooc.git
 * [new branch]      master -> master
[ubuntu@ip-172-31-34-196:~/bitstarter]$
```



Figure 30: Once we determine that staging is ready to ship, we simply `git checkout master`; `git merge staging`; `git push production-heroku master:master` to change to the master branch, merge in the commits from staging, and push to the production-heroku remote. At this point the site is deployed at the production Heroku URL.

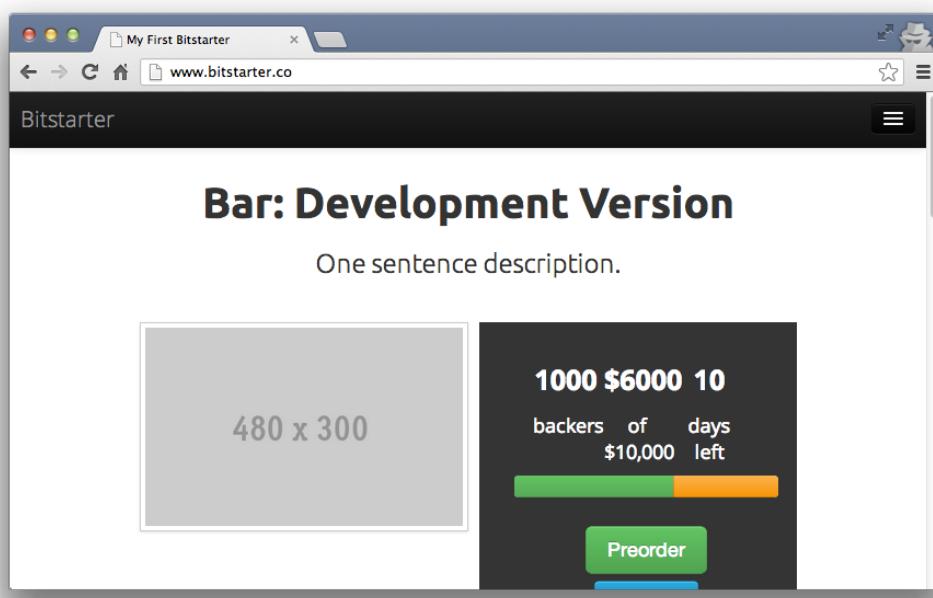


Figure 31: Once we have deployed to production, the very last step is viewing the same content via our custom domain name. To learn how to set this up, read the next section.

DNS and Custom Domains

This section is optional but highly recommended. It is optional as registering a domain will cost a little bit of money (<\$20-\$30), but will allow you to use your own custom domain (`example.com`) rather than a `example.herokuapp.com` domain. But this is recommended as a good domain will certainly help in raising money for your crowdfunder.

DNS Basics

Before registering our custom domain, let's talk briefly about the Domain Name System (DNS). When you type a domain name like `example.com` into the URL bar of a browser and hit enter, the browser first checks the local DNS cache for the corresponding IP address. If missing, it sends a request to a remote machine called a *DNS server* (Figure 32), which either gives back the IP address (e.g. 171.61.42.47) currently set for the domain name or else asks another DNS server nearby (Figure 33) if it knows the IP address. Given an IP address, your browser can then attempt to establish a connection to that IP address using the [TCP/IP protocol](#), with the [HTTP protocol](#) then being a layer on top of that.

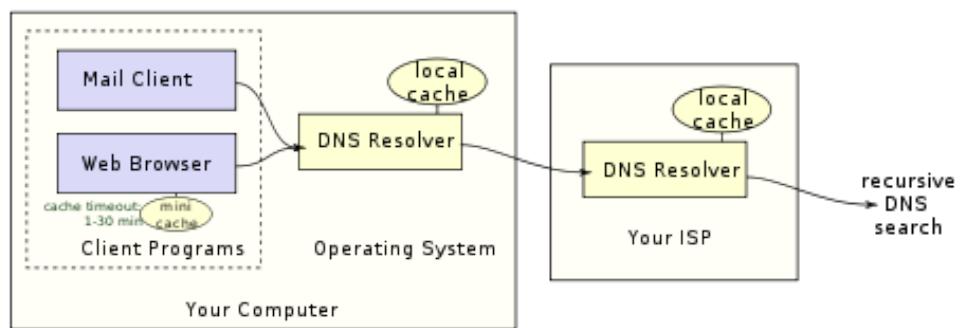


Figure 32: When you input a domain name into your browser's URL bar, your computer first tries to look up the information locally, then remotely at the ISP, and then through recursive DNS search (Figure credit: Wikipedia).

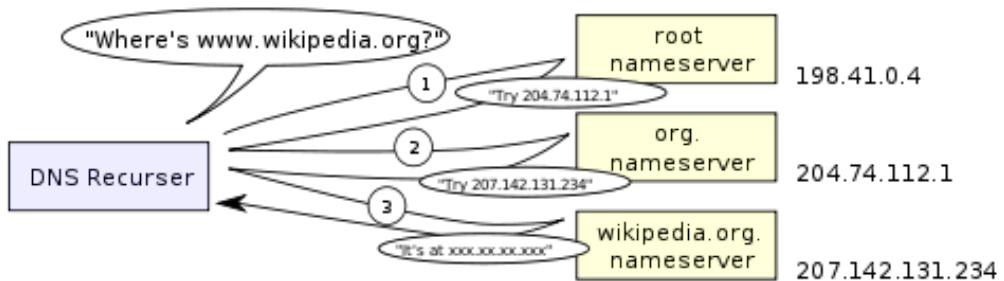


Figure 33: This is what the recursive resolution mechanism for DNS looks like. The end result is the IP address of the requested URL. (Figure credit: Wikipedia).

So there are at least three protocols at play here ([DNS](#), [TCP/IP](#), and [HTTP](#)), which seems surprisingly complicated. You might reasonably ask why we even need DNS at all, and why we don't just type in IP addresses directly into the browser. This actually does work (Figure 34), but has two caveats. First, humans find it much easier to remember words than numbers, which is why we use domain names rather than IP addresses for lookup. Second, engineers have utilized this "bug" of having to do a DNS lookup and turned it into a "feature"; some sites will have multiple servers with different IPs behind a given domain name (e.g. for [load balancing](#)), and on these different IPs may get repurposed for other tasks. The point is that the IP address behind a domain name can change without notice, especially for larger sites, so the domain name is really the canonical address of a site rather than the IP address.

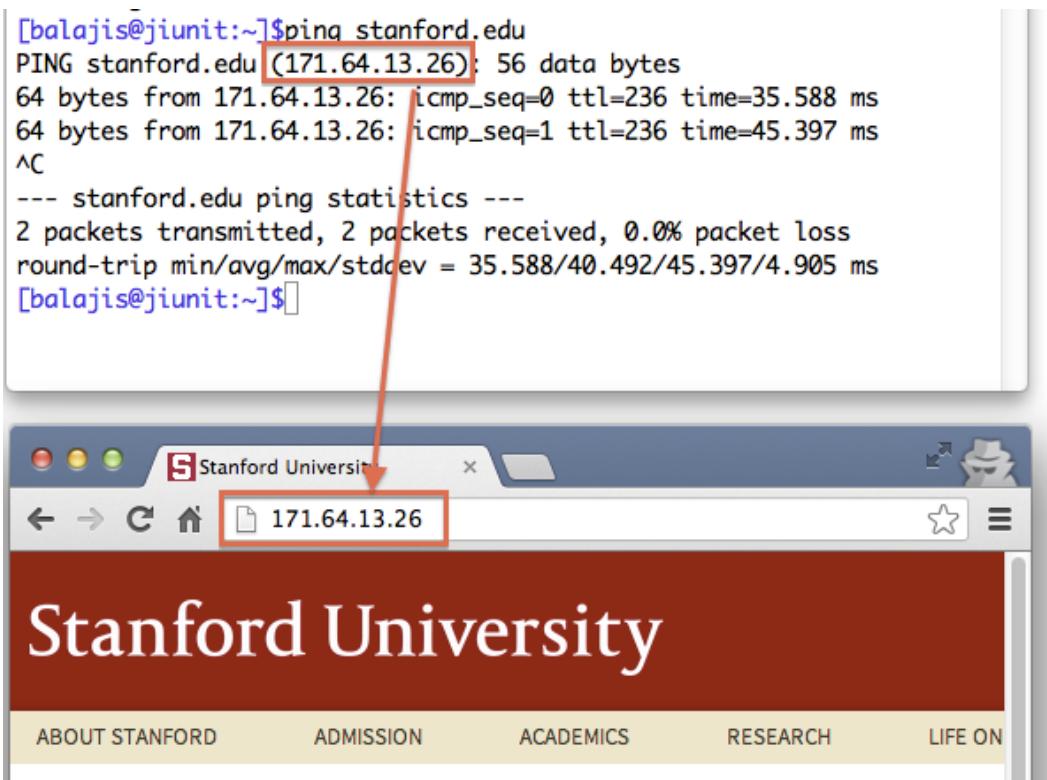


Figure 34: You can use `ping` or another command to find the IP address associated with a domain and navigate to it directly. However, the exact IP address for the remote server is subject to change, while the domain will likely stay constant.

While DNS is a [distributed database](#) in the sense that many nodes have a copy (or partial copy) of the domain-name-to-IP lookup table, it is highly nondistributed in one crucial sense: who is allowed to write new domain names to the database. For the last twenty years² the domain name registration system has been [centralized](#), such that one has to pay one of many licensed registrars to get a domain name.

²With the development of Bitcoin it is [now possible](#) to distribute many protocols that were previously thought to be fundamentally reliant upon a central authority. In particular, the [Namecoin project](#) is a first stab at implementing a distributed DNS; it may not end up being the actual winner, but it has reawakened research in this area.

Finding a good domain: domize.com

When choosing a domain name, keep the following points in mind:

1. It should be easily pronounceable over the phone or in person.
2. If it's a product, consider having a unique string that distinguishes it; this will make it easier to find when just starting out (e.g. `lockitron.com` or `airbnb.com` were fairly unique).
3. You should consider SEO/SEM, using the Google Keyword Planner and Facebook Census tools from the market research lecture to determine the likely traffic to the domain once it can rank highly in Google search. **Exact matches** for search queries are highly privileged by Google, in that `ketodiet.com` will rank much more highly than `ketosis-dietary.com` for the search query [keto diet].
4. It should be as short as possible. If the `.com` is not available, get the `.co` or the (more expensive) `.io`. That is, get `example.io` over a spammy sounding name like `my-awesome-example.com`. The expense of the `.io` domain here is a feature rather than a bug, in that it disincentivizes domain squatters.
5. If you use a non `.com` domain, consider using one that Google has whitelisted as a **Generic TLD**, unless you are genuinely international and want to rank for non-English searches (in which case, say, an `.fr` domain may be appropriate).
6. See whether you can get the corresponding Facebook, Twitter, and Github URLs (e.g. `facebook.com/ketodiet`).
7. For the US: see whether you can get the trademark using trademark search. A tool like this may be available in your country as well.
8. Much more difficult: see whether your name has a negative connotation in another language. It's almost impossible to check this without knowing many languages right now (though you could imagine creating a naming webservice that helped with this), but we mention it for completeness.
9. Don't spend too much time on a name, or too much money on a premium domain name from a place like `sedo.com` unless it's absolutely crucial for your business. Dropbox used `getdropbox.com` for many years before getting the `dropbox.com` URL, and FB used `thefacebook.com` before getting `facebook.com` and eventually `fb.com`. Though these did become nontrivial inconveniences it certainly didn't kill them. The only difference: rather than `getexample.com` or `theexample.com`, today one might do `example.io` first, and then get the `.com` later.

The `domize.com` site is a very useful tool for going through many combinations of domain names (Figure 35-36).

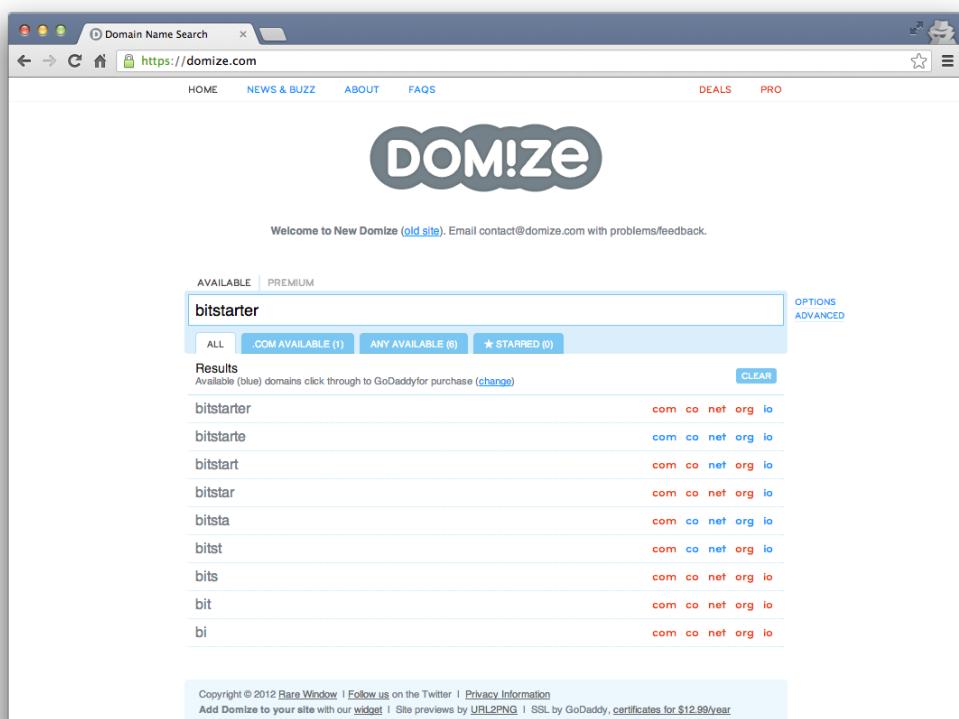


Figure 35: Use domize to find an available custom domain name.

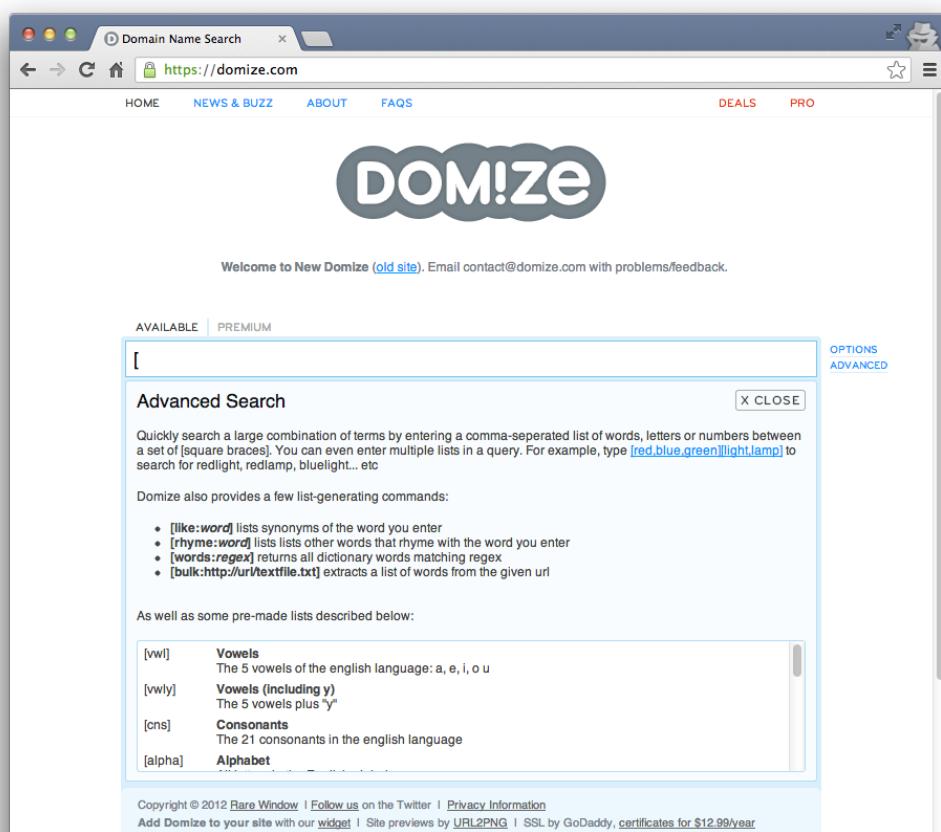


Figure 36: Domize supports many advanced features for finding domains. Play around with these.

Registering a domain: dnsimple.com

Once you've found an available domain through domize.com you will want to register it. Though it is more expensive than pure registrars like GoDaddy or NameCheap (say \$20-\$30 vs. \$6), DNSimple is a good choice for a single domain which you plan to register and utilize *immediately*. This is because it gives you an extremely well designed UI and API for working with the DNS records for your domain. This is particularly helpful for beginners as DNS can be quite confusing. In Figures 37-39 we show how to sign up for DNSimple, register a domain, and edit DNS entries.

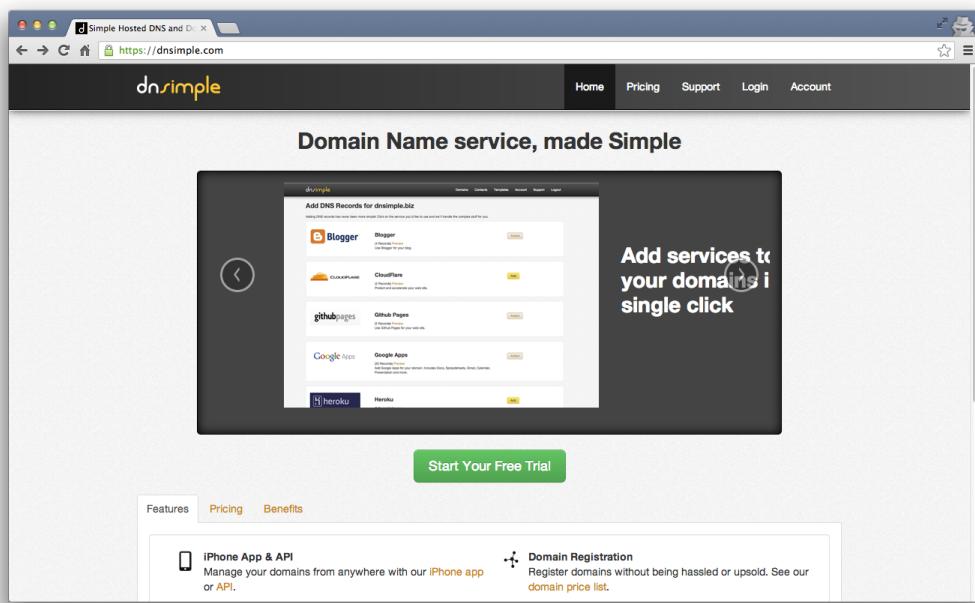


Figure 37: Sign up for dnsimple.com.

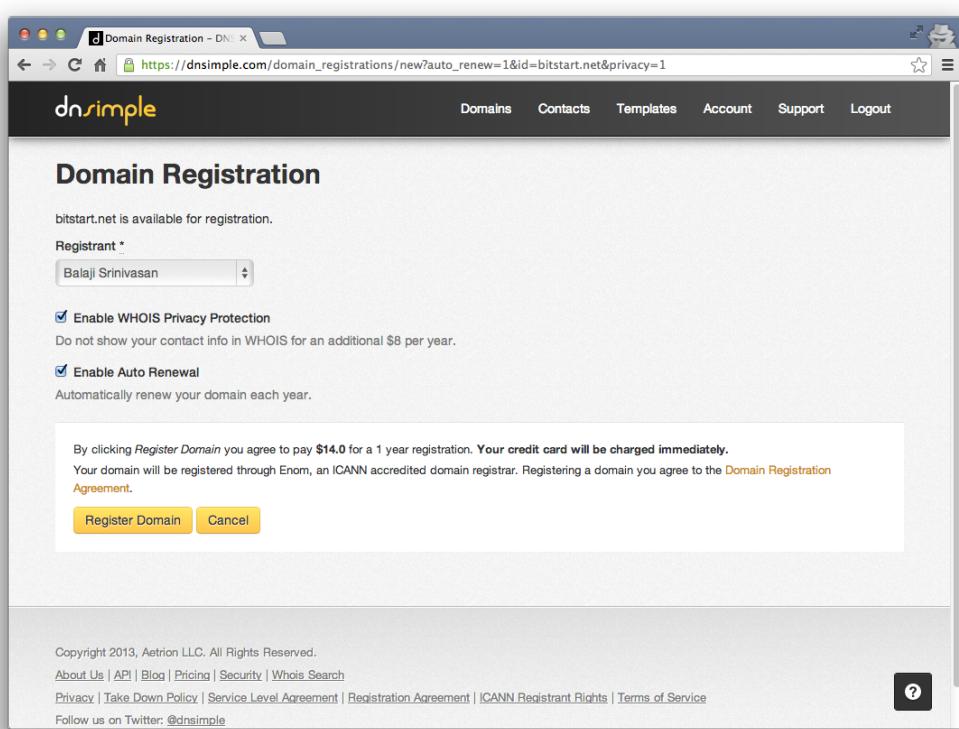


Figure 38: At dnsimple.com/domains/new, you can register a domain. NOTE: you WILL be charged if you do this, so only do it if you actually want a custom domain. We do recommend enabling WHOIS protection to prevent your address being emailed by spammers.

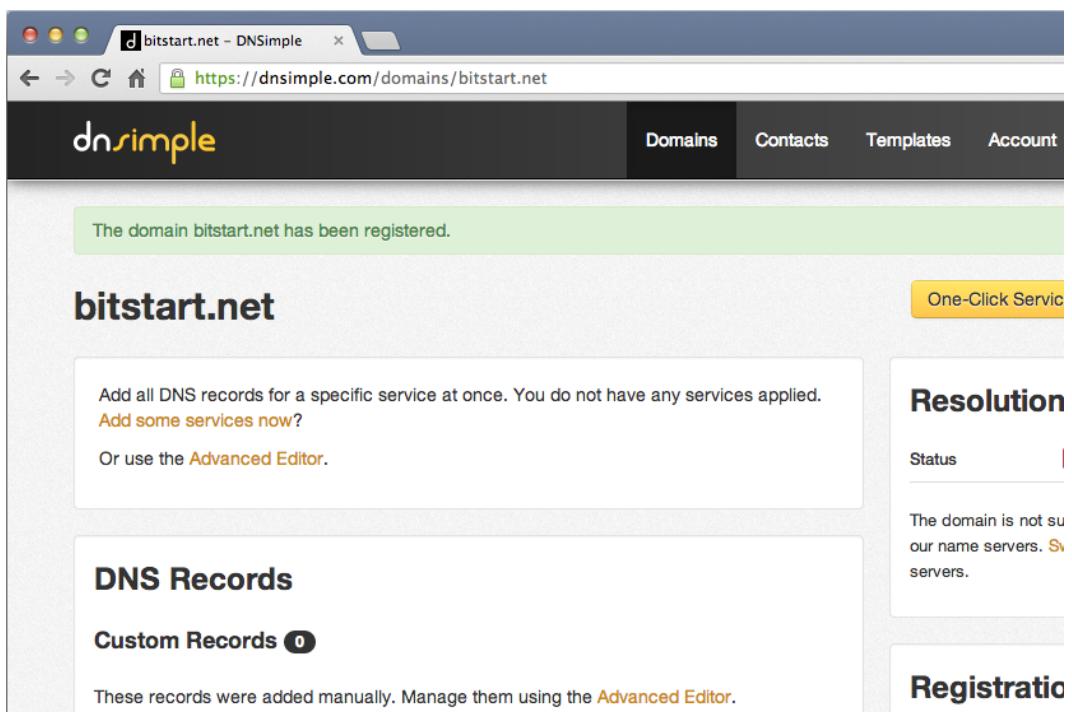


Figure 39: Now you have a web interface for managing your domain.

Configuring your DNS to work with Heroku

So, at this point you used domize.com to pick out a custom domain `example.com` and dnsimple.com to register it. You also have³ `example-site-staging.herokuapp.com` and `example-site.herokuapp.com` live. But how do we make the `.com` point to `example-site.herokuapp.com`? We need to do two things. First, we need to configure things on the DNS side; with DNSimple this is very easy. Next, we need to tell Heroku to accept requests from this new domain. Finally, we confirm that it works by viewing things in the browser. Read the instructions [here](#) and [here](#). Then look at the screenshots below.

The screenshot shows the DNSimple web interface for the domain `bitstarter.co`. The URL in the browser bar is `https://dnsimple.com/domains/bitstarter.co`. The main page displays the domain name `bitstarter.co` prominently. A callout box highlights the text "Add all DNS records for a specific service at once. You do not have any services applied. Add some services now?" with an arrow pointing to the "Add some services now?" button. To the right, there's a "Resolution Status" section showing "Status: Resolving". Below that is a "Registration Status" section with details: Status: Registered, Expiration: Feb 28, 2014, Privacy: Feb 28, 2014, and a list of tasks: Change Name Servers, Renew Domain, Make Public, Renew Whois Privacy, and Disable Auto-Renewal. On the left, there are sections for "DNS Records" (Custom Records 0, System Records 5) and "SSL Certificates" (No SSL certificates for this domain). At the bottom right, there's a "Tasks" section with one item: Change Contacts.

Figure 40: Your goal is to have end users see a custom domain like `bitstarter.co` rather than `bitstarter.herokuapp.com`. To do this, first go to `dnsimple.com/domains/example.com`, plugging in your own registered domain for `example.com`, and then click `Add some services now` as shown.

³Note that in this case the custom domain (`example.com`) has a similar name to the Heroku site(s) (`example-site-staging.herokuapp.com`, `example-site.herokuapp.com`). But this is by no means necessary, and can be useful to have staging URLs which do not include the custom domain's name (e.g. `foo-bar-baz.herokuapp.com`).

https://dnsimple.com/domains/bitstarter.co/applied_services

dnsimple

Domains Contacts Templates Account Support Logout

Add DNS Records for bitstarter.co

Adding DNS records has never been more simple! Click on the service you'd like to use and we'll handle the complex stuff for you.

Blogger (4 Records) [Preview](#) Use Blogger for your blog. [Add](#)

CloudFlare (2 Records) [Preview](#) Protect and accelerate your web site. [Add](#)

djee.se (2 Records) [Preview](#) The easiest way to build websites with django CMS. [Add](#)

githubpages (2 Records) [Preview](#) Use Github Pages for your web site. [Add](#)

Google Apps (20 Records) [Preview](#) Add Google Apps for your domain. Includes Docs, Spreadsheets, Gmail, Calendar, Presentation and more. [Add](#)

heroku (2 Records) [Preview](#) Use Heroku as your web host. [Add](#)

Figure 41: Click Add as shown.

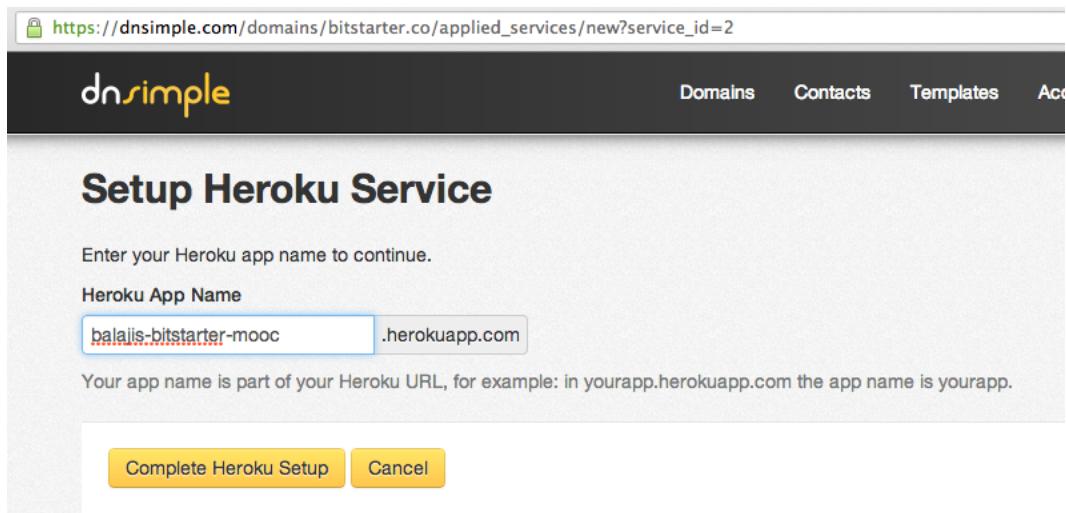


Figure 42: Enter your app name. Note that this is your *PRODUCTION* app name (e.g. `balajis-bitstarter-mooc.herokuapp.com`) not your *STAGING* app name (e.g. `balajis-bitstarter-s-mooc.herokuapp.com`). This is because an external domain is only important for end users; the staging URL is viewed only internally by devs.

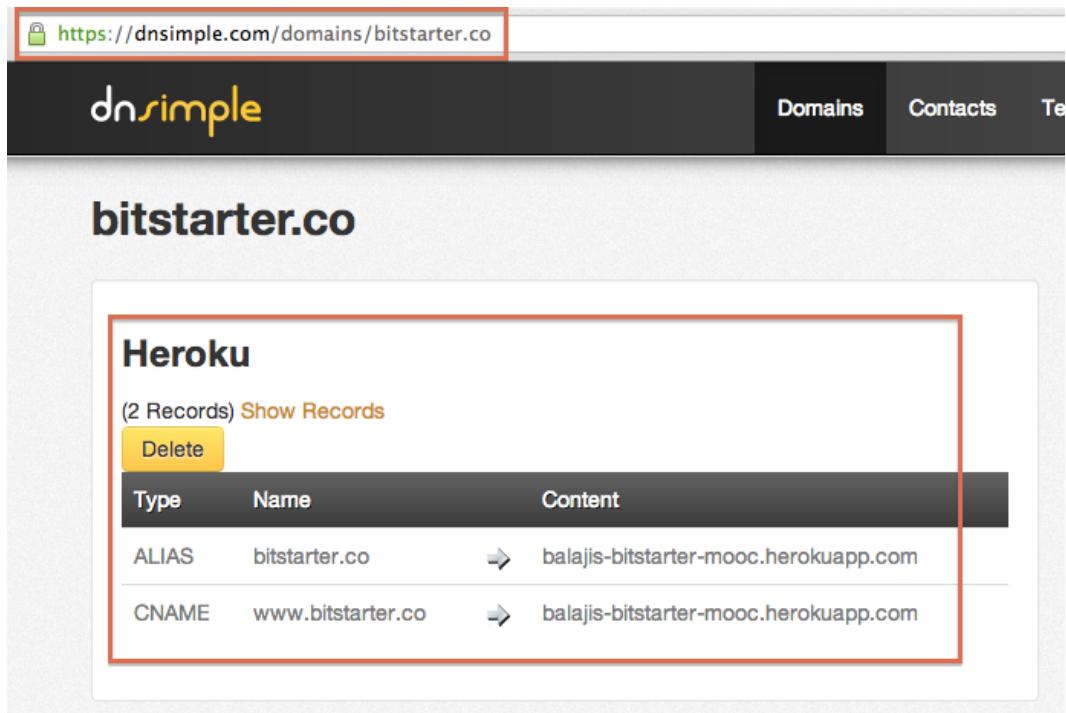


Figure 43: Once things are set up, then you should see the following on your DNSimple homepage for this domain.

```
[ubuntu@ip-172-31-42-185:~/bitstarter]$heroku domains:add --app balajis-bitstarter-mooc www.bitstarter.co
Adding www.bitstarter.co to balajis-bitstarter-mooc... done
[ubuntu@ip-172-31-42-185:~/bitstarter]$heroku domains:add --app balajis-bitstarter-mooc bitstarter.co
Adding bitstarter.co to balajis-bitstarter-mooc... done
[ubuntu@ip-172-31-42-185:~/bitstarter]$
ip-172-31-42-185 0$ emacs 1$* bash 2-$ bash
```

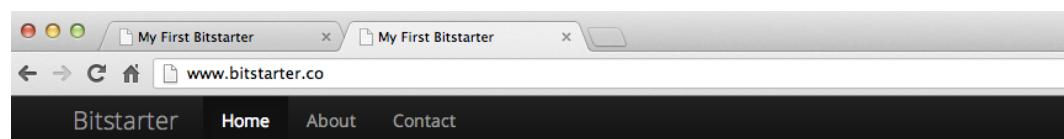
Figure 44: Your final and key step is to authorize Heroku at the command line to accept connections to that app from both the `www.example.com` and `example.com` variants.



Product: After Staging

One sentence description.

Figure 45: If all goes well, this works in the browser.



Product: After Staging

One sentence description.

Figure 46: If you have set it up right, both the `www.example.com` subdomain and `example.com` should route to the same place.

Congratulations. You just set up a custom domain.

Setting up HTTPS and Google Apps

These final two sections are also optional, but recommended if you really want to build out a site.

1. *Get an SSL certificate.* This is a bit expensive (up to \$100 if you get a wildcard certificate), but required to permit logins or other authenticated HTTPS connections to your website. To do this you can buy an SSL certificate through DNSimple at <https://dnsimple.com/domains/example.com/certificates/new> (substituting your own domain name). Then configure your SSL setup with Heroku, as detailed in these instructions: [1](#), [2](#), [3](#). Unless cost is a very pressing issue, get a wildcard certificate as it'll make your life easier in the long run (more details: [1](#), [2](#)).
2. *Set up Google Apps for your domain.* You will probably want to be able to receive email at your domain to process feedback during the crowdfunder. One way to do this is by setting up Google Apps for your domain. Note that you might be able to get a free account through [this link](#), but otherwise [Google Apps costs](#) about \$50 per user per year (quite inexpensive as business software goes). If you want to continue, first [sign up for Google Apps](#), configuring any desired aliases (e.g. `admin@example.com` and `press@example.com`), and then [set things up within DNSimple](#) to support Google Apps mail, chat, and other services. Then wait a little while for DNS propagation and then test it out. This process will be fastest on a new domain; with domains bought from others it can take up to one week for all the DNS records to propagate over.

If you do both of these steps, you should be able to both support HTTPS connections and be able to set up arbitrary email addresses at your new custom domain.

Mobile

The Age of Internetification

The basic assumption behind mobile computing is that we are living through an age of internetification, similar to the age of electrification in the early 20th century. Over the span of several decades, electricity evolved from a [World's Fair curiosity](#), to a [sensation](#) in which electrical engineers were the computer scientists of their day, and ultimately to a [utility](#) whose ubiquity and [simple UI](#) belies the stunning [complexity](#) of the electrical grid. We don't think about it much today, but our world became *electrical*: every device that could conceivably benefit from electrical power became powered by electricity. Edison's electric light bulb [goosed](#) the buildup of the electrical grid, and it was soon used to power [irons](#), [sewing machines](#), and (over time) [factories](#).

In the same way, today we are seeing internet connectivity evolve from flaky 56k modems towards the idea of [wireless broadband](#) as a *utility* whose presence you can increasingly simply assume. Just like the process of electrification packaged up an enormously complicated grid into the trivial UI of a wall plug, we've likewise started to move from wrestling with [Trumpet WinSock configuration](#) to the trivial UI of an [Airplane Mode](#) switch. And with internetification our world will become *mobile*: every device that can conceivably benefit from an internet connection will be connected to the internet. Jobs' internet-connected smartphone goosed the buildup of the mobile internet, but it is now being tasked with connecting [locks](#) and [thermometers](#) and [mines](#).

The long-term logic of internetification is inescapable, which is why mobile computing is not just a *trend*, but the future of computing (Figures 1, 2, 3). Mobile web applications in particular promise to leave behind desktop web applications in the same way that desktop web apps left behind native desktop apps, and for similar reasons. The fundamental user and developer conveniences afforded by an application downloaded from the internet and continuously updated (i.e. a webapp) generally [trumped](#) the fancier widgets that desktop apps provided. Similarly, the sheer convenience of having an app that is constantly in your hands at all times - and that can either change its state based on your location or else make your physical location entirely irrelevant - will increasingly trump the fancier widgets and increased screen real estate afforded by desktop web apps (1, 2, 3, 4, 5).

Smartphone Usage = Still Early Stage With Tremendous (3-4x) Upside

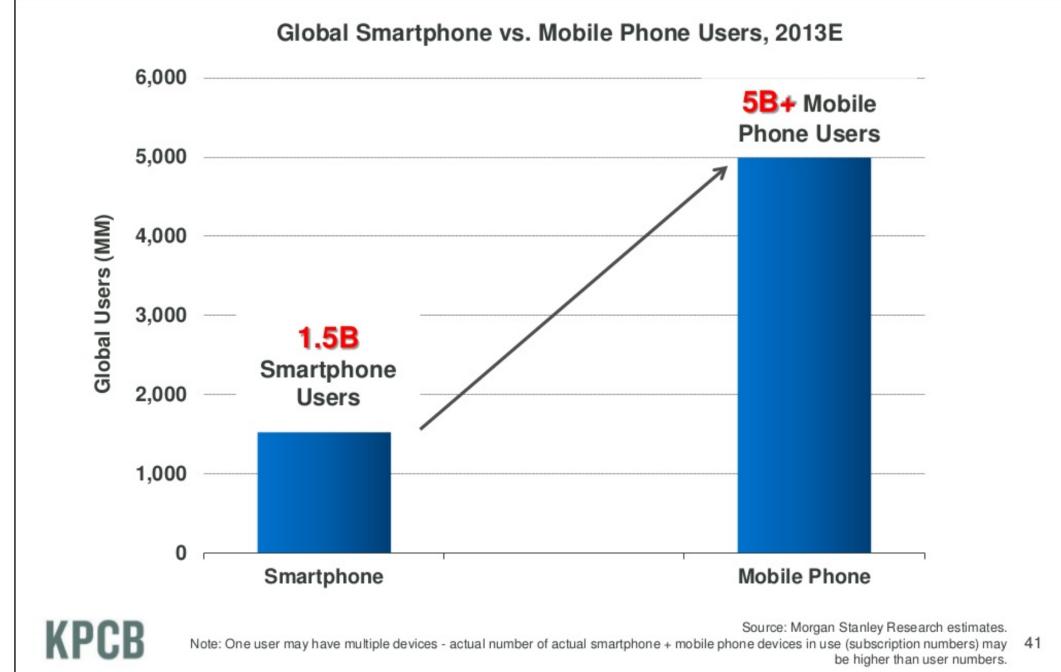


Figure 1: With an estimated world population of 7e9, global mobile penetration is roughly 70% for all phones and 20% for smartphones (Source: [Internet Trends](#) by Mary Meeker at KPCB, who attributes the statistic to Morgan Stanley Equity Research).

Smartphone Subscriber Growth = Remains Rapid 1.5B Subscribers, 31% Growth, 21% Penetration in 2013E

Rank	Country	2013E Smartphone Subs (MM)	Smartphone as % of Total Subs	Smartphone Sub Y/Y Growth	Rank	Country	2013E Smartphone Subs (MM)	Smartphone as % of Total Subs	Smartphone Sub Y/Y Growth
1	China	354	29%	31%	16	Spain	20	33%	14%
2	USA	219	58	28	17	Philippines	19	18	34
3	Japan*	94	76	15	18	Canada	19	63	21
4	Brazil	70	23	28	19	Thailand	18	21	30
5	India	67	6	52	20	Turkey	17	24	30
6	UK	43	53	22	21	Argentina	15	25	37
7	Korea	38	67	18	22	Malaysia	15	35	19
8	Indonesia	36	11	34	23	South Africa	14	20	26
9	France	33	46	27	24	Netherlands	12	58	27
10	Germany	32	29	29	25	Taiwan	12	37	60
11	Russia	30	12	38	26	Poland	11	20	25
12	Mexico	21	19	43	27	Iran	10	10	40
13	Saudi Arabia	21	38	36	28	Egypt	10	10	34
14	Italy	21	23	25	29	Sweden	9	60	16
15	Australia	20	60	27	30	Hong Kong	8	59	31

2013E Global Smartphone Stats: Subscribers = 1,492MM Penetration = 21% Growth = 31%



Note: *Japan data per Morgan Stanley Research estimate. Source: Informa. 40

Figure 2: Detailed breakdown of smartphone users by geography. (Source: [Internet Trends](#) by Mary Meeker at KPCB).

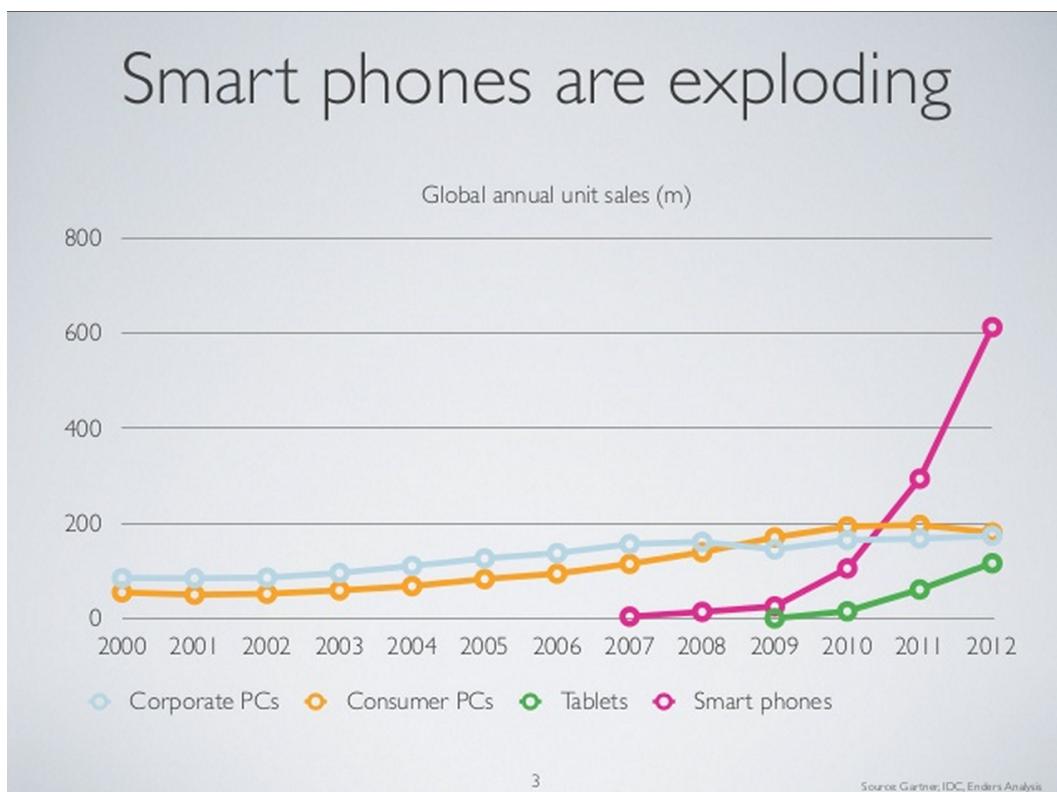


Figure 3: The installed base of smartphones and tablets has roared past that of PCs. (Source: Benedict Evans, BEA May 2013).

Is Mobile Simply a Fad?

There are certainly technologies that were in retrospect dead ends or fads; [XHTML 2](#) and (more arguably¹) [Second Life](#) fall into this category. And you will hear programmers of a certain stripe complain about how HTML5 and other browser technologies are just reinventing the same tools that were present on the desktop 15 years ago. “Wow”, they will say - “now I can rotate a sphere in a [browser window](#) rather than a [desktop window!](#)” While true, this minimizes the [novel aspects](#) of web apps, prime among them being user convenience. Web apps require zero installation, are always up to date, have a familiar browser interface, can be integrated with other apps, and (if well-designed) have bookmarkable and shareable URLs. Putting [Quake in a browser](#) with zero software to install results in a qualitatively different app, if only for the [rapidity](#) of viral spread; the time delay between starting a game and pasting a link to said game in an email or Reddit thread is much shorter than the time to get ten friends to buy and install a desktop application.

Similar complaints are often voiced regarding mobile apps, as many mobile apps are indeed simply stripped down desktop apps, with fewer features and often no obvious reason to be mobile - aside from convenience. But when it comes to the marketplace, that convenience is everything. Here’s Yishan Wong, Reddit CEO and former Facebook Director of Engineering on [the topic](#):

Why is mobile the future?

Q: So all I hear is that the mobile internet, mobile devices, geolocation etc, is the future. The web, and the web on desktops is dead. When I ask people why, they say ‘coupons’. You check in at a bar and then send you coupons. A coupon driven revolution or paradigm shift seems like weak-sauce to me. Can someone tell me why mobile is the future without mentioning coupons. Or will the next revolution really be based around deals and monetary incentives?

Yishan Wong: The answer to this so simple that it boggles the mind.

The reason is that human beings are mobile.

We are not sessile organisms confined to a single location for all our lives. If you look at every other technology ever developed, you’ll see that the ability to carry it around and use it in arbitrary locations vastly increases its utility and versatility.

Therefore, the emergence of (1) sufficiently powerful computing processors to run a client device and (2) always-on low-latency global coverage to link these clients to an arbitrary cloud computing and storage resource means that computing power is now mobile. Computers are an incredibly useful tool, but now that you can carry a powerful one around with you, thus removing the constraint that you sit in a single location if you want to do any computing and furthermore, extending the available things in the world upon which you can apply computation.

¹This is arguable because it is possible that [Oculus Rift](#) may be to Second Life what the [iPhone](#) was to Pandora: a new technology that gives a second shot in the arm to an idea that was [too early](#).

So: the reason isn't "coupons" at all - not even close. Coupons is one extremely minor and narrowly specific application of computing mobility. The real reason mobile is "the future" is much bigger and abstract: every tool of value to humans is and always was mobile to begin with, and if it had to start out in its early phases as something large and cumbersome, that's merely a temporary state upon which there exists a natural force that would immediately make it mobile if only it could be made lighter and more portable.

The Mobile Present

We should make this clear at the outset: for the purposes of the *class*, we'll be building a basic crowdfunding webapp which uses Bootstrap's responsive features to get a reasonable level of mobile-friendliness. However, for the purposes of your longer term business you may indeed want to build a native app. So it's worth surveying the state of mobile computing today.

Google/Samsung/Apple and Android/iOS

We begin with the obvious: the dominant players are [Samsung](#) and Apple (on the hardware end) and Google's Android and Apple's iOS (on the software end). After them, Amazon's [Kindle Fire](#) tablet is worthy of mention, as is the hotly anticipated [Firefox Phone](#). Of these players, iOS is still on top from a monetization standpoint, as reflected in the fact that as of about a year ago, YC startups [still developed](#) for iOS first. Here's Paul Graham on the topic:

To most startups we fund, iOS is way more important. Nearly all build for iOS first and then maybe one day port to Android. There are a few exceptions like [Kyte](#) who use Android to do things you can't do on iOS. And of course [Apportable](#) has been very successful auto-porting iOS apps to Android.

However, while Apple has cumulatively paid developers more than [\\$10 billion](#) over the five year lifetime of iOS, Google is gaining major ground. Android has long been the frontrunner in the mobile space by handset volume, is about to catch up on [downloads](#), and has closed the app revenue revenue share gap from 81%/19% to 73%/27% in the [span of a year](#). Android is also starting to roll out new web services (e.g. [gesture typing](#), [Google Now](#)) that iOS can't seem to [match](#). While cliché to state it, with Jobs' passing it is likely that the long-term game is going to Google.

It's also important to know how these players make their money. Apple makes about [\\$1B](#) in profit annually from the App Store while Google did [\\$17.5M in May](#) on \$350M in revenue (\$210M profit on \$4.2B revenue annualized). However, Google's real mobile monetization is through [mobile ads](#) (roughly \$8B annual revenue as of 2013) and Apple's real monetization is through first party hardware sales (roughly [\\$160B annualized](#) revenue and \$40B profit as of Q2 2013, of which the majority comes from iPhone and iPad sales). Samsung is actually the party which is making money from Android hardware sales, toting up an [impressive](#) \$8.3B in profit on \$50B in revenue in Q2 2013 alone.

Thus right now the revenue from the mobile hardware buildout currently leads that from the software buildout by a significant margin. It's almost tautological that this is the case -

one needs devices in people's hands before one can install software - but this lag still means significant opportunity. For example, Kleiner announced a [fund](#) of \$200M for iPhone software when the App Store launched in 2008; that looked aggressive at the time, but they may make that back on [Square alone](#), alongside their other mobile bets on Shopify, Path, and Flipboard (all much-needed wins after a decade of [cleantech losses](#)).

Mobile Cannibalizes Hardware while Growing Software

Perhaps the best way to conceptualize why the mobile hardware buildout is epochal for the software industry is that hundreds of millions of new customers bought handheld computers while they thought were buying phones, thereby vastly increasing the theoretical market size for internet-connected computers and applications. Moreover, in addition to simply being a mobile computer, smartphones in particular can be thought of as replacing (Figure 4) a wide swath of specialized devices (1, 2, 3, 4, 5, 6), and enabling new modes of user interaction: the phone as a remote, the phone as a [docked device](#), the phone as a port for more specialized sensors or devices.

The combination of these trends means trouble for traditional purveyors of standalone consumer electronics hardware (e.g. Garmin, Sony) and tremendously more reach and distribution for software developers. An increasing number of hardware device equivalents (Figure 4) are now available at all times, and interesting apps like Square and Clinkle combine these built-in hardware APIs in clever ways. Square used the fact that the iPhone headphone jack was actually an [active port](#) that received a signal input from the headphone jack. They used this to make a hardware device that plugged into the headphone jack and sent credit card information over this low-bandwidth channel, thereby creating a cross-device front-facing credit card scanner, avoiding Apple's [tax](#) on devices that use the Dock Connector, and spawning a company with a [\\$3.25B](#) valuation. Clinkle hasn't yet been released, but [appears](#) to use a smartphone's built-in microphone and speaker as an analog communications channel, thereby sending payment information via ultrasound encoding without the necessity of an internet intermediary. But one doesn't need this level of cleverness to do something of general utility: as a software engineer, the smartphone buildout means you can assume the hardware will simply be an API call away.



Figure 4: A mobile phone replaces a slew of specialized devices. (Source: [devost.net](#)).

Mobile Constrains Business Strategy

If you are launching a software-based startup today, it is important to think about whether there is *any* way you can put a fundamentally mobile app (native or web) at the core of your service. Those businesses that did not think about this angle up front have been forced to adapt. Apple was one of the very first to see this: they [changed their name](#) from Apple Computer right before the launch of the iPhone, to redefine themselves as a mobile devices company. At a smaller scale [Taskrabbit](#) (a very useful service for finding temporary labor) has been forced to pivot to compete with Exec, which targeted the same market but focused on a mobile/realtme experience (i.e. get assistance via phone from anywhere with rapid response time). Pandora similarly struggled for years to achieve revenue until the iPhone arrived and [supplied](#) the missing distribution they'd needed all along. As Tim Westergren, Pandora CEO [noted at the time](#): “The iPhone changed everything for us. It almost doubled our growth rate overnight. More importantly, it changed way people think of Pandora, from a computer service to a mobile service.”

To incorporate mobile into your strategic thinking may seem like a trendy, artificial constraint. It's obvious that mobile apps are useful for [transportation](#), [tourism](#), [maps](#), and anything related to a change in location. Yet what does mobile have to do after all with (say) business expenses or furniture shopping? However, then you start thinking “oh, I'll turn the phone into a mobile scanner for business receipts” and you get [Expensify](#). Or you might reason that “it would be useful to barcode-scan an arbitrary object with your phone to comparison shop” and you get [Amazon Flow](#), which retailers complain is turning their shops into [glorified showrooms](#).

The reason this “think mobile” constraint is actually useful is that if you [can](#) figure out a fundamentally mobile angle for your app, you will have come up with something new that was technically infeasible five years ago (before the launch of the iOS App Store) and practically infeasible until relatively recently (now that ubiquitous smartphone penetration in many countries can be assumed). As such your app will² invalidate the assumptions of incumbents in the space, granting you a competitive advantage.

The Mobile Future

ChromeOS and Mobile HTML5

There is good reason to believe that [Mobile HTML5](#) is going to wax in importance for app development. The reason is that Google's [Sundar Pichai](#) now runs Google Apps, Chrome, and Android, after taking over from Android founder Andy Rubin. Google has always thought of native apps as a sort of near-term distraction forced on it by the unexpected success³ of Apple's iOS. In this vision, HTML5/ChromeOS was a bet on the future and Android was a bet on the present. But the replacement of Rubin with Pichai seems to be an indication that HTML5 is now Google's present, rather than its future. With Apple [stumbling](#) and Android gaining, the serious threat from iOS appears to be losing momentum and with it

²As [Sun Tzu](#) noted, “Thus the highest form of generalship is to balk the enemy's plans; the next best is to prevent the junction of the enemy's forces; the next in order is to attack the enemy's army in the field; and the worst policy of all is to besiege walled cities.”. In other words, to win against an opponent with billions of dollars, *always attack assumptions*.

³Indeed, even Apple was surprised by this. Few recall this today, but at the launch of the iPhone in 2007 Jobs recommended that people write [web applications for mobile Safari](#). It took a year for them to respond to overwhelming developer demand and build the App Store and the [iOS SDK](#).

the “distraction” of native. Perhaps the most likely scenario over the next 12-24 months is that Pichai will merge Android into ChromeOS ([0](#), [1](#), [2](#), [3](#), [4](#), [5](#)), likely by doing backwards-compatible rewrites of many Android Java API calls in Javascript. The best analogy for this is MacOS’ [Carbon to Cocoa](#) transition in the early 2000s. And this means that if you want to play the *really* long game, you might want to start looking at [Google Drive Apps](#), the [Chrome Web Store](#), and the [Chromebook Pixel](#).

The Internet of Things

The so-called internet-of-things (Figure 5) takes mobile even further, moving from items like smartphones or tablets equipped with touchscreens to items which may run mostly unattended and/or be designed to be controlled from a nearby phone's mobile programmable screen. Note that to produce such items again presumes ubiquitous (a) wireless broadband ("internetification") and/or (b) handheld computers as a basic assumption. Another basic premise behind the internet-of-things is that every device will ultimately have an IPv6 address or the functional⁴ equivalent. The first internet-of-things devices are already on the market, including [Nest](#) (thermometers), [Lockitron](#) (locks), and [PayByPhone](#) (parking meters); see this [post](#) and [this image](#).

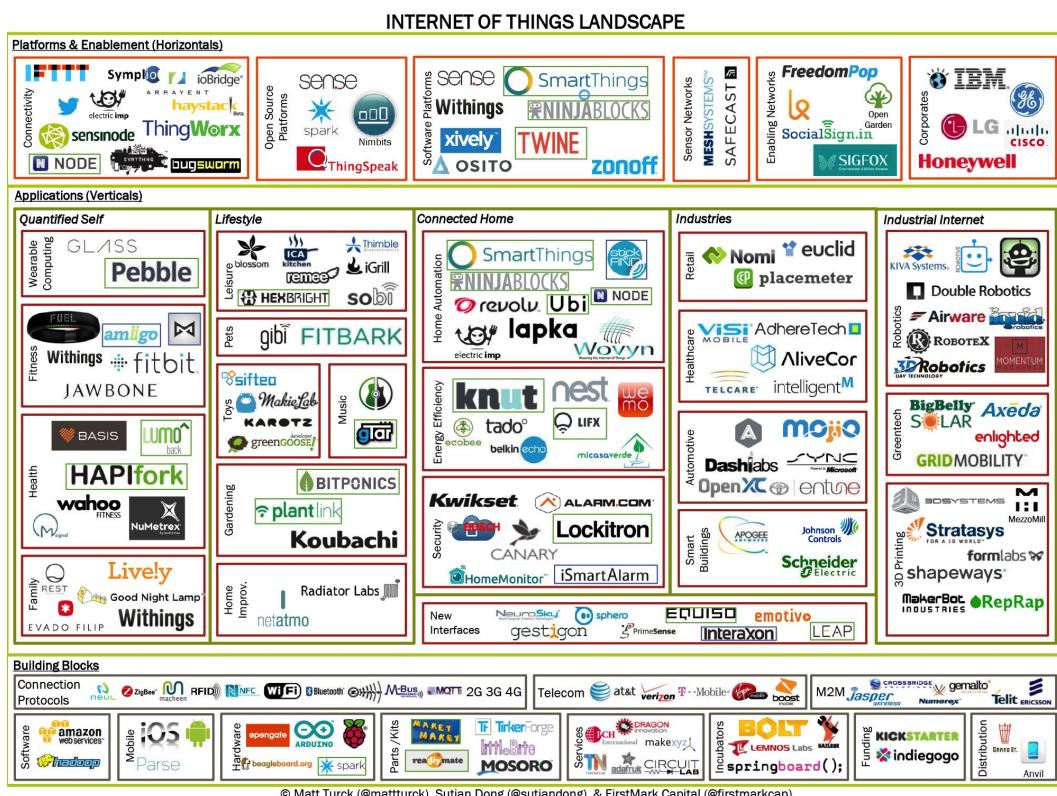


Figure 5: An overview of companies involved in the internet of things. (Source: Matt Turck, TechCrunch).

⁴If you are interested in this area, read up on IPv6, Carrier-grade NAT (CGN), and the tradeoffs between the two. In normal circumstances one might predict that providers don't have major short-term incentives to pursue IPv6 rather than CGN. However, due to Google Fiber pushing IPv6, it's a reasonable bet that IPv6 will happen over the resistance of some ISPs.

Quantified Self

Closely related to the internet-of-things is the concept of *quantified self* or QSelf (1, 2, 3, 4, 5). The motivation is that today one can more easily determine what is happening in Bangalore or Budapest than what is happening in one's own body. We can change this by developing sophisticated sensors for the human body that feed data locally into mobile phones or directly to a remote database over the internet. Products in this category include the [Fitbit](#) and [Jawbone Up](#) armbands, the [Withings scale](#), and the [Scanadu Scout](#) vital signs sensor.

While currently quite modest in market size, QSelf technology is likely to ultimately disrupt the way medicine is practiced. As motivation for this statement, consider the contrast between the healthcare industry and the fitness industry. Doctors routinely state that “if fitness were a drug, it would be prescribed for everyone”. Study after study rolls in on the benefits of personal fitness; the effect sizes are off the charts (1, 2, 3, 4, 5). Yet your fitness is considered to be your own responsibility: you can join gyms or get good personal trainers, but ultimately the buck stops with you - you need to take the initiative. But consider how that initiative plays out in healthcare. If you come to a doctor’s appointment wanting to talk about something you’ve researched, doctors generally get vexed. Either you are right (and thereby undermining their authority) or you are wrong (and thereby dismissed). This state of affairs is odd: you are with your body for a lifetime, whereas the doctor is only with you for twenty minutes each year. Thus the one area of medicine that really works - fitness - operates quite differently from the rest of medicine in practice.

The fundamental difference between the two areas is the presence/absence of mandated intermediaries. You do not need to [pay](#) a physician, an insurance company, a hospital, or ([indirectly](#)) a federal regulator in order to step on a scale. But you do need to make some or all of these payments in order to get a prescription medical test. QSelf is going to change all of this, by making many body sensors fundamentally mobile and thereby making it impractical to centrally limit access to diagnostics. We’re already seeing this to some extent; for example, a sequel to the Fitbit sleep monitor could enable [at-home](#) sleep apnea tests, and the [Scanadu Scout](#) or an equivalent will likely [replace](#) an expensive trip to the doctor’s office to be examined with a stethoscope and a blood pressure cuff. Ultimately this particular mobile technology path culminates in vastly improved self-diagnostic capabilities. And from there? Perhaps even a movement to [legalize](#) self-treatment.

Mobile Technology

Let’s now introduce two concepts which will be useful for understanding how we can handle the profusion of mobile devices: HTTP User Agents and CSS Media Queries.

Preliminaries: HTTP and User Agents

You’ve surely heard of the Hypertext Transmission Protocol (HTTP), which is how browsers talk to web servers (among other things). This protocol is a complex beast that deserves its own class, and we won’t get into the guts of it here. The main thing you need to know for now is that when you connect via browser to a remote web server, you are sending an HTTP Request and getting back an HTTP Response. An [HTTP Request](#) can be thought of as a function call (GET, POST, etc.) that accepts a positional argument (a resource like a URL) and a long list of optional keyword arguments (the HTTP Headers). You can generate HTTP requests with:

- A command line tool (ex. curl)
- A web browser (ex: Chrome)
- A library call (ex: node's restler)
- A web UI (ex: hurl.it)

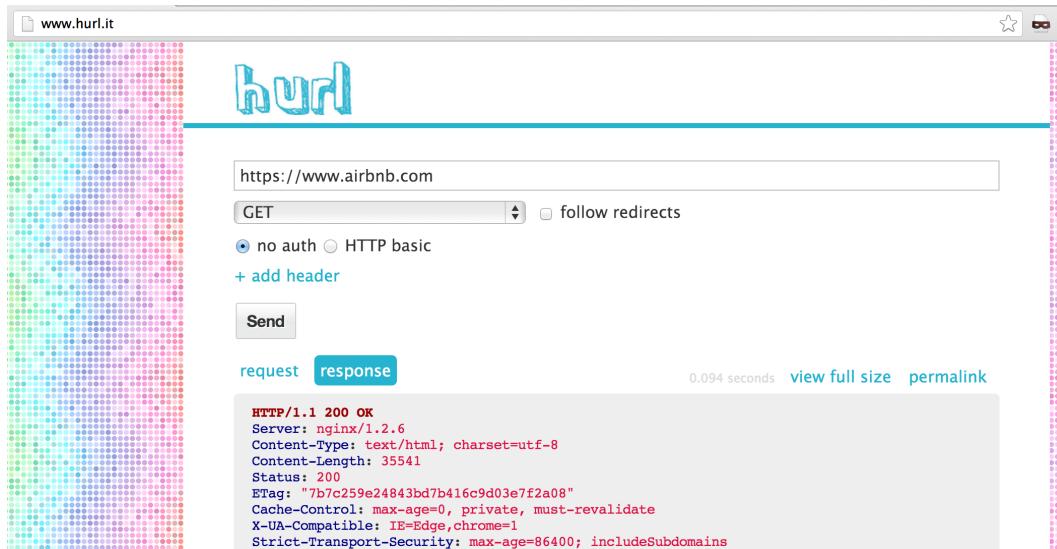


Figure 6: Here's an example of an HTTP GET request to the resource <https://www.airbnb.com>.

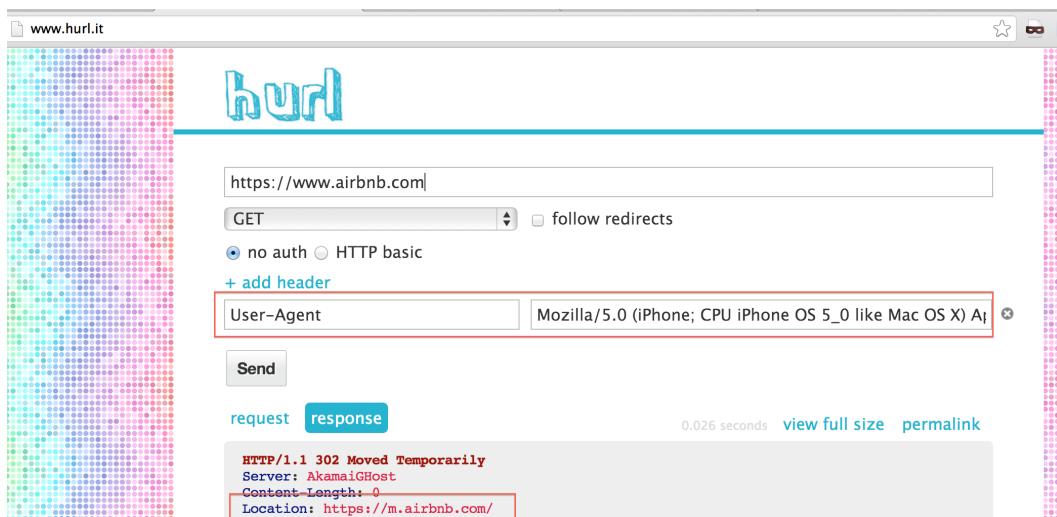


Figure 7: And here's that same HTTP GET request to the resource <https://www.airbnb.com>, except with the *iPhone 5 User Agent*. Note that the HTTP response is different.

When a web server receives an HTTP request it issues an [HTTP response](#). The exact response can depend upon the time of the day, the user's location, or many other parameters - including the value of the HTTP request's [User-Agent](#) header (Figures 6-7). The [User-Agent](#) (UA) is the string that the HTTP client presents as self-identification (examples). It is trivial to spoof the [User-Agent](#) and thereby pretend that you are a device different from what you are (e.g. Chrome's [User-Agent Switcher](#)). Nevertheless, on the server side one can do so-called "User Agent sniffing" as a first cut approach to determine what device is connecting to our site. While no-doubt [fraught with issues](#) related to the sheer profusion and unreliability of UA strings, this technique is nevertheless used by many major websites (e.g. AirBnB) to serve up different content for different user devices. We should note that many references counsel against UA sniffing and recommend feature detection instead, using a library like [modernizr](#) ([1](#), [2](#), [3](#), [4](#)). Sometimes feature detection is indeed what you want, e.g. if determining whether the given client supports a particular HTML5 API call, but often that can turn into a bit of a Rube Goldbergish way to simply specify that you have a custom site for (say) iPhones or iPads. Use your judgment here.

Preliminaries: CSS Media Queries and Responsive Web Design

We talked about Cascading Style Sheets (CSS) a bit in Lecture 6. In Homework 4, CSS is used not just to control typography, for layout, and for graphics, but also to detect the screen width and use this to apply styles [conditionally](#). This is called a *media query*; it's basically a rudimentary if/then statement which executes different CSS code based on the width of the current browser window. Media queries are the basis for Responsive Web Design (RWD), where the page layout adapts to the screen width of the viewing client. See examples [here](#).

The Mobile Problem: Diversity of Devices

With these preliminaries, we can discuss an important issue in "going mobile": namely the multiplicity of screen widths (Figure 8) and OS versions on Android (Figure 9), and more generally the wide array of potential API clients (Table 1).

Table 1: *The general mobile API client fragmentation problem. From the point of view of API-based design or service-oriented architecture (SOA), mobile means not just mobile devices, but the set of potential clients/distribution targets for your application/content. Note that for several of these targets (iPhone, iPad, Android) one may also need to design an alternate layout of the page for portrait and landscape mode.*

API Client	Client language/libraries
Desktop web	HTML/CSS/JS
Mobile web	HTML/CSS/JS
iPhone	Objective-C/iOS
iPad	Objective-C/iOS
Android devices	Java/Android
Native Mac app	Objective-C/Cocoa
Native Windows app	HTML/CSS/JS/Metro UI or XAML
Command line	Node, Python, Ruby, Java, ...
3rd party API client	Node, Python, Ruby, Java, ...
Twitter cards	Twitter API

Continued on next page

Table 1: The general mobile API client fragmentation problem. From the point of view of API-based design or service-oriented architecture (SOA), mobile means not just mobile devices, but the set of potential clients/distribution targets for your application/content. Note that for several of these targets (iPhone, iPad, Android) one may also need to design an alternate layout of the page for portrait and landscape mode.

API Client	Client language/libraries
Facebook Open Graph	FB API
Google Rich Snippets	Google Microformats
Google Glass	Glass Cloud API
iWatch (?)	Likely Objective-C/iOS
iTV (?)	Likely Objective-C/iOS

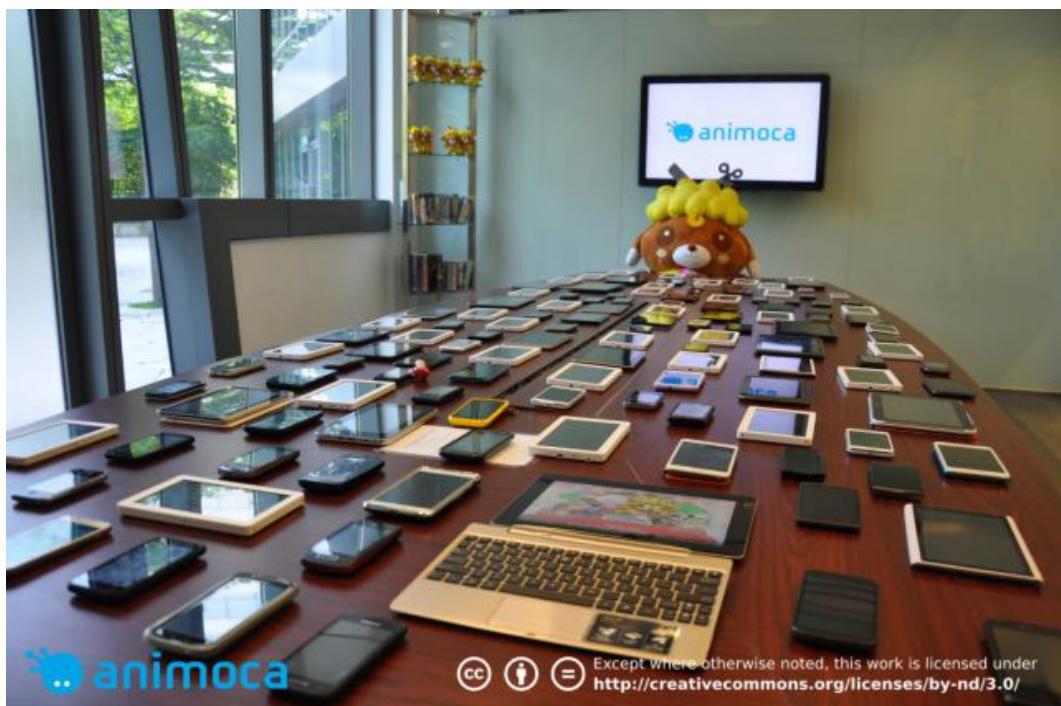


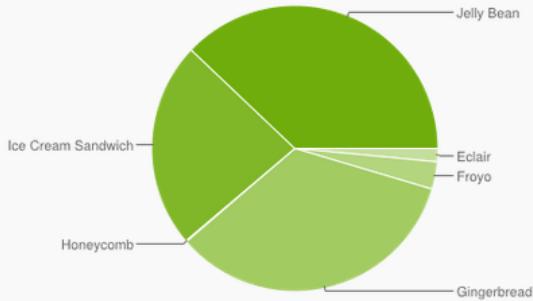
Figure 8: The Android OS screen fragmentation problem. (Source: [Techcrunch](#)).

Platform Versions

This section provides data about the relative number of devices running a given version of the Android platform.

For information about how to target your application to devices based on platform version, read [Supporting Different Platform Versions](#).

Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.4%
2.2	Froyo	8	3.1%
2.3.3 - 2.3.7	Gingerbread	10	34.1%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	23.3%
4.1.x	Jelly Bean	16	32.3%
4.2.x		17	5.6%



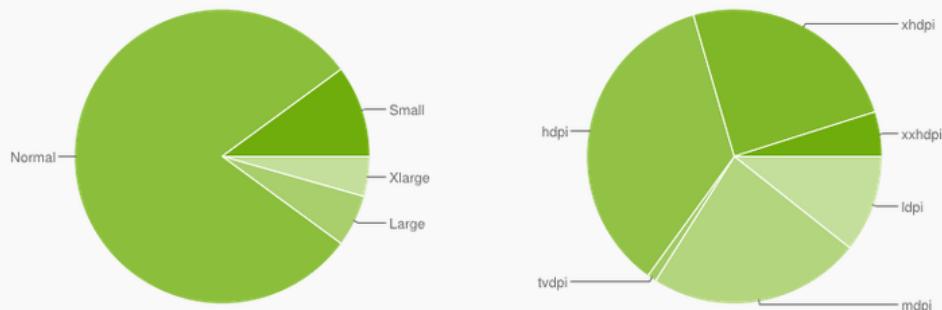
*Data collected during a 14-day period ending on July 8, 2013.
Any versions with less than 0.1% distribution are not shown.*

Screen Sizes and Densities

This section provides data about the relative number of devices that have a particular screen configuration, defined by a combination of screen size and density. To simplify the way that you design your user interfaces for different screen configurations, Android divides the range of actual screen sizes and densities into several buckets as expressed by the table below.

For information about how you can support multiple screen configurations in your application, read [Supporting Multiple Screens](#).

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	9.9%			0.1%			10.0%
Normal	0.1%	16.0%		34.9%	24.0%	4.9%	79.9%
Large	0.6%	3.2%	1.0%	0.4%	0.5%		5.7%
Xlarge		4.1%		0.2%	0.1%		4.4%
Total	10.6%	23.3%	1.0%	35.6%	24.6%	4.9%	



*Data collected during a 14-day period ending on July 8, 2013.
Any screen configurations with less than 0.1% distribution are not shown.*

Figure 9: The Android OS fragmentation problem (Source: [Google](#).)

While a bit of a pain, the situation is not as complicated as these figures and tables may make it out to be. From a business perspective, you need to consider whether you will be optimizing for sheer global availability (in which case a mobile web app with RWD is the first choice), for maximum current App Store revenue (in which case iOS-first is still the best), or for the maximum number of handset users and potential future app revenue (in which case Android-first may be worth a bet).

From a technical perspective, there are essentially two options for dealing with client diversity in the context of mobile web apps. At one extreme one can use the one-size-fits-all approach of responsive web design. In this case, the server returns the same HTTP response for all clients, with CSS media queries in the body of the response used to implement some [conditional logic](#) (e.g. hiding or showing certain divs on a phone or tablet). At the other extreme the server can return *different* HTTP responses for different User Agents, as specified in the headers of the HTTP request. This can be used to deliver custom mobile-optimized sites and is what Facebook, Google, AirBnb, Twitter, The Guardian, TechCrunch, and now [Github](#) are doing (often via a URL like `m.example.com`). An even more labor-intensive version of this is to build both a mobile web app and a native app for each client.

The second approach of custom sites for each client is significantly more expensive, but can be partially facilitated with a so-called API-based design or [Service-Oriented Architecture](#) (SOA). In this setup the core of the application is an API calling a database. All clients (internal command line tools, the mobile web, and iOS/Android) then simply call this API to create/read/update/delete objects and then render them using the tools of their native environment: e.g. ASCII for the command line, HTML for the mobile web, iOS widgets for iOS apps, and so on. (Table 1) That said, even with this kind of approach one needs engineers to constantly maintain the templates for each and every device (and in both landscape and portrait mode), ensuring that they don't break or subtly lose functionality as operating systems are updated and new devices come out.

While Facebook can afford this kind of investment, for a small company it is infeasible to support many different clients. And it is usually unnecessary: picking one platform and using it to prove out your market is probably the best idea. Steve Jobs' heuristic was to pick a "technology in its spring", a technology which is fresh and growing yet has been used in production applications by at least a few people. One way to measure this is by comparative Github forks/stars, Stackoverflow activity, Google trends/search data and the like. Obviously, the ideal first platform will change over time; for example, Instagram timed their pure iOS bet perfectly, but a company that was doing something similar today might seriously consider doing Android first (or perhaps even getting a Google Glass dev kit).

Mobile Frameworks: Pure HTML5 vs. Cross-Compilation

Let's make one point of clarification up front, regarding the distinction between "libraries" and "frameworks". While a little fuzzy, the difference is that a library is built to solve a single problem. You just `import` or `require` it and then use it in conjunction with several other libraries. By contrast, a framework is a collection of libraries with its own internal data structures and conventions that ends up governing the way you approach a space. When it comes to web development, the advantage of using a set of libraries is that you have great flexibility; the disadvantage is that in filling the gaps between these libraries you will end up writing your own framework. Conversely, the advantage of a framework is that it's "batteries included", but the disadvantage is that it can be hard to do something that greatly violates

the underlying assumptions of the framework.

For the purposes of the class we'll be using the [Bootstrap 2](#) framework, because it's widely used, gets you on base quickly on all platforms, and is very flexible and skinnable as frameworks go (e.g. [wrapbootstrap.com](#) and [bootswatch.com](#)). Moreover, by using Bootstrap 2 it'll be relatively easy to migrate your app to the new [Bootstrap 3](#), which is now a true mobile-first framework. Two possible alternatives to Bootstrap 3 are [JQuery Mobile](#) (JQM) and [Sencha Touch](#) (ST). Both of these are pure mobile-optimized HTML/CSS/JS frameworks which don't require knowledge of iOS and Android development. The sites built with these are still viewable in a desktop browser but are optimized for the mobile web ([example](#)). With some effort you can usually mix and match components from JQM or ST into a primarily Bootstrap app (or vice versa), but it's generally [easier](#) to just stick with one⁵ of these frameworks.

One might also consider using some of the cross-platform frameworks like [Phone Gap](#) or [Appcelerator Titanium](#). These frameworks let you write an app in HTML/CSS/JS and then cross-compile it to native Android and iOS apps (and [also](#) to Windows Phone, Blackberry, etc.). While a valiant effort in many respects, these kinds of frameworks are in our view a bit of a no man's land between a pure web app vs. a pure native app. An RWD app with Bootstrap 2 will get you "on base" on each platform with something serviceable, while a pure native app will get you high performance, access to the hardware, easier debugging, and native-looking widgets. But a cross-compiler framework is yet another thing to learn, and won't usually give you the same snappiness of a native app. It's arguable as to whether it's really better than (1) just biting the bullet and going native or (2) alternatively, taking your initial RWD web app and making a mobile-optimized 2.0 version.

Now, the choice between these two is the question of whether or not you can get a mobile HTML5 app up to the same level of performance/snappiness as a native app, which is perhaps the most important strategic debate in mobile development today. Facebook famously tried and [failed](#) to do a pure HTML5 mobile app, at least by their standards and with their approach. However, this [impressive demo/post](#) by Sencha Touch does indicate that significant performance improvements over FB's initial version are feasible. Still, LinkedIn [recently announced](#) that they also switched back from mobile HTML5 to native due to memory issues and widget snappiness. In general it is fair to say that it is still nontrivial to get a mobile web app up to the same level of snappiness as a native app, but at the same time a mobile web app gives you a reasonable first presence on every device and is quick to build.

A Mobile Solution: RWD first, Native immediately after, and then use logs

To recap: we're targeting the mobile web for this class because (a) it is relatively easy and will get you "on base" for each platform and (b) it will be useful no matter what you do. However, it should be explicitly noted that even if ChromeOS and mobile HTML5 are the future, the present is still very much native. If you really do want to build a mobile app, you'll want to learn either [Android](#) or [iOS](#) well, with the mobile web app as an initial stopgap. That said,

⁵Note however that Bootstrap 2 would be best considered a CSS framework, and is meant to work with JQuery (a JS library). You can also easily combine Bootstrap 2 with a frontend JS framework (like [Backbone](#) or [Angular](#)), and a backend web framework (like [express](#) for node.js). That is, Bootstrap focuses on the CSS styling of widgets as opposed to their behavior and how they move data around the page (frontend JS) or back and forth from the server (backend framework). By contrast, JQuery Mobile and Sencha Touch have opinions about how data is moved around the page and maintained within widgets (see e.g. the JQuery Mobile [AJAX section](#)). Does all this sound complicated? Yes, welcome to the fast-moving world of web development. You may enjoy [this rant](#).

here is a reasonable approach to get started with an app as of mid-2013.

1. Start with a responsive web design app, built on top of an API.
2. Don't try to build an RWD framework yourself; this is an [enormous effort](#) in its own right. Use Bootstrap 2 (or if you want to be aggressive, the [pre-release Bootstrap 3](#)). You might also consider Zurb Foundation.
3. Theme your Bootstrap app with an off-the-shelf theme from wrapbootstrap or bootswatch or the like. Customize with Google Fonts or get a designer from dribbble.com or 99Designs.com to do more in-depth customization.
4. Now deploy your site and get some users.
5. Analyze your [log data](#) very carefully, looking at [User-Agent](#) in particular to determine what devices your customers are using.

This approach will let you up and running quickly with something which is functional on virtually every platform. Then you can use the log data with your business strategy to prioritize further targets. Alternatively, you can launch the RWD app and then go with (say) an iOS or Android app, using the log data to prioritize your second target (depending on your product, that might even be [IE clients](#) rather than iOS clients). The use of an [API at the core](#) of the site will then allow you to generalize to new platforms over time (e.g. Google Glass or Apple's Watch, as well as embedding your app in Twitter or FB).

Mobile Constraints

It's finally worth noting that there are a variety of new issues that crop up with mobile.

- *The network is unreliable.* While we're moving towards an age of ubiquitous wireless internet, we aren't fully there yet. Apps should be designed to work with flaky connections or at least diagnose and display that they are offline. (See: "[The Fallacies of Distributed Computing](#)")
- *File sizes need to be small.* Once you get things working, you'll want to start aggressively removing unused JS functions and CSS styles from what you're serving to mobile clients as they are often on low bandwidth connections (e.g. 3G or 4G rather than Wifi).
- *Debugging requires logging.* You want to log aggressively and provide user bug reporting. The nature of mobile device diversity is that you will not be able to anticipate up front all the odd issues that will arise in the field. Do make sure to get your users' permission to send these log files back to your server periodically.
- *Minimize user input.* You should use passcodes over passwords, swipes over typing, and autocomplete over form filling. A great example of this is the [Google Maps autocomplete field](#). It replaces 5-6 fields (Address 1, Address 2, City, State, Zip, Country) and includes error checking to boot.
- *Minimize time to result.* Your app should boot quickly (difficult with an RWD app) and offer big buttons with common use cases.

In some ways the constraints of a mobile environment - where every byte matters, and network connections can be as frustrating as 56k modems - warp us back to 1995 when the internet was young. But bear in mind that was a time of great opportunity.

Social/Local/Mobile, Virality, and Growth

Overview

Social, Local, and Mobile. The mantra of venture capitalists from roughly¹ 2007-2012. It's a **catchphrase**, a **buzzword**, and arguably a **cliche** in Silicon Valley. The term and its relatives attract disparagement because they are so closely associated with photosharing, social gaming, check-ins, and recreational apps of this nature. Now, there's actually much that is admirable about the companies in these areas: Instagram is a highly complex engineering **feat**, Snapchat does **account** for a significant fraction of all photos taken in the world, and Zynga **was**, at one point at least, a multibillion dollar public company². Still, these are with some justification seen as cotton candy apps, as the kind of apps that healthy twenty-somethings **make for each other**, rather than apps that solve problems of real significance. But is that really all there is to say about "social, local, and mobile", though? Let's see if we can dive into the phenomenon term by term to understand why VCs love it and whether there's something deeper there.

Mobile

Let's begin with mobile. We've already discussed this in some detail, but to get some scale of the buildout, let's begin from the [June 2007](#) launch of Apple's iPhone. Since that time point approximately **1.5 billion people** - fully 20% of the world's population - have purchased smartphones of some kind (either from Apple, Samsung/Google, or another vendor). Let's say that's been six years and two months, or 74 months. A simple division gives us the average number of smartphones bought per day over this time period:

$$\frac{1.5 \times 10^9}{74 \times 30.5} \approx 6.65 \times 10^5$$

So more than half a million people have bought a smartphone every day for the last six years in a row. And it's still growing, as at least 3.5 billion more will be installed to catch up to world feature phone penetration. It's also one with heavy turnover, as phones are replaced every few years with new models bristling with new sensors. Additionally, mobile app stores (the App Store and Google Play) provide unprecedented distribution comparable only to the World

¹"SoLoMo" hasn't exactly died off, but the latest mantra is **consumerization of the enterprise**. A genuine trend, a real thing enabled by the proliferation of mobile devices over the last few years and the rise of business social networks like LinkedIn, Salesforce Chatter, and Yammer - but somewhat faddish nonetheless.

²Yes, you might want to do something that's more technologically interesting. But once you start a company and are involved with operational details, you quickly realize that great drama and effort lies behind even the most innocuous pastel-colored divs. In Hollywood it's often said that the serious movies tend to be comedies behind the scenes, while the comedies tend to be deadly serious when the cameras stop rolling. It's thus a significant mistake to assume that company which is fun and games on the surface is significantly easier to build from an operational perspective than a more obvious technology play like SpaceX. As proof: **Facebook** and **Snapchat**'s founder imbroglios, Instagram's **antitrust** holdup, and Zynga's **shareholder lawsuit**. Startups that involve large dollar figures become stressful, serious business quickly, even if their external appearance is fun and games.

Wide Web itself, and combine it with the crucial aspect of monetization. And tens of billions of dollars in profit are being made every quarter in this space. So of these three buzzwords, mobile is certainly not overhyped. VCs love mobile because it offers an enormous, rapidly growing market for software entrepreneurs along with unprecedented monetizable distribution for apps.

Social

And what of social? Though also legitimately used on a [regular basis](#) by one-billion-plus people, the social arena is older³ and less profitable than mobile, due to the lack of physical hardware sales comparable to iOS/Samsung. Yet let's give credit where it's due; social networking is influential enough to spark [civil wars](#) and occupy a [significant portion](#) of the industrialized world's conscious attention. Major players in this space include the usual suspects (Facebook, Twitter, LinkedIn, Google+), the newer entrants (Pinterest, Instagram, Path, Quora, Tumblr), and the older-but-still-fairly-popular crowd (Flickr, Reddit). Of all categories, we assume these companies to be the most familiar and won't spend much time reviewing them.

Like mobile, the buildout of social networking is impressive. Facebook achieved [one billion](#) monthly active users (MAUs) in approximately eight and a half years of operation, corresponding to the following average user installation rate:

$$\frac{1 \times 10^9}{8.5 \times 365} = 322,320$$

At 86,400 seconds per day, that means signing up roughly four people per second, every second, for eight years straight. Not bad. Incredibly impressive in fact, but this also puts mobile's growth into stark relief. To sign up for a social network one pays nothing and need only click a link. To obtain a mobile phone one must travel to a store and pay hundreds of dollars. Yet the installation rate of smartphones significantly exceeds even the extraordinary growth of social networking, and might well be one of the [fastest growing](#) technologies of all time. This gives some sense of the relative utility of the two areas, and why Facebook has been so interested in getting into mobile, now with considerable [success](#). We'll return to perhaps the most important aspect of social networking in a bit (namely the concept of *virality*), but it's worth pausing to consider two points.

First, a social connection is much more permanent and well-maintained than an email address. This may not seem like that significant an issue till we do a brief calculation. Let's say that you have 730 friends, and a given friend changes their location, their email address, or their phone number every two years on average. Then if you were manually keeping an address book on pen and paper, you'd have to make an edit every day. And this assumes that said friend broadcasts their information out to all of their friends every time these values are updated. Instead, today we just keep a [pointer](#) to someone's Facebook or LinkedIn profile; in this manner changes to their contact information are automatically available to all their contacts. Indeed, this insight was behind Sean Parker's second company, [Plaxo](#); part of his reason for joining Facebook was that it was going after the same thing in much greater generality and with much more success.

³You can date it all the way back to Six Degrees in 1997, but Friendster in 2002 was probably the first breakout social network. See this [IEEE timeline](#).

Second, if one thinks of the internet as a quasi-frontier, one can think of social network connections as digital *roads* between people, as channels over which one can send digital packages. Today they’re used to send photos, videos, and chat messages. And social networks are mocked for this, for the idea that we set up the internet simply to share cat videos. Tomorrow, though, these roads may be carrying digital payloads with significantly more significance, like [source code](#) or [3D printer schematics](#) or Bitcoin. As every good becomes digital, a new or existing social network has the potential to become a *trade route* and not just a communication channel, with [Gumroad](#) as one of the first along these lines but with many more (likely using the internet-native Bitcoin currency) to follow. At that point social starts to get even more interesting.

Virality

Let’s now return to the topic of virality. As we noted above, Facebook’s signup form *alone* received approximately⁴ four submissions per second for the last eight and a half years, a significant load for any web application. And yet how many times do people sign up for accounts compared to their use of the service? If you consider that most users signed up once or twice within the past eight years, but have executed tens of thousands of writes and reads on the [facebook.com](#) domain over that period, one starts to get a sense of the sheer colossal scale of the site. How did Facebook achieve such scale? In a word, *virality*. We illustrate the importance of virality with this cautionary tale of Hipstamatic vs. Instagram (Part 1, 2, 3):

Hipstamatic was one of the first startups to crack the photo formula in the mobile space—then it watched similar services gain ground and eventually blaze by. The company’s experience proves that no startup can rest on its laurels in the age of the iPhone, when the time between innovation and disruption is ever shortening, and when IPOs and fast exits are valued over establishing long-term viable businesses. And perhaps most significantly, Hipstamatic proves that no modern startup can ignore the siren call of social, even if at its own peril. . . .

In October 2010, Hipstamatic was booming. Its business model of selling in-app digital lenses and films, which effectively turned your iPhone into an old-school Polaroid camera, was attracting millions of users and millions of dollars in revenue, especially from its fast-growing community of shutterbugs in industries ranging from fashion to media. . . .

So on Oct. 6, when an ex-Goolger named Kevin Systrom launched a photo-sharing service called Instagram, there was no way of knowing that it would mark the beginning of the end of Hipstamatic’s honeymoon. Like Hipstamatic, the iPhone app enabled users to add vintage-era filters to photographs, but there were two key differences: Instagram was free and inherently social; Hipstamatic was not.

. . .

By March of 2011, when Hipstamatic hired its new designer, Laura Polkus, Instagram had already rocketed to 2.2 million users, and was growing by 130,000 users per week. But Polkus says the team largely ignored Instagram. “There wasn’t a

⁴ And these are users that log in at least every month, so-called Monthly Active Users (MAUs) as distinct from those that log in at least every day Daily Active Users (DAUs). If we include every signup of a fake account, or an account that just isn’t checked that often, we’ll get much larger numbers.

whole lot of attention paid there,” says Polkus, who was later let go. “The conversation internally was, “Well, we’re completely different. They are a social network, and we are not. Who cares what’s going on with them? We’ll just continue to do what we do.’ But from the public’s perspective, that’s obviously not the way things were seen.”

“As Instagram started to build, everyone was like, ‘You guys should do this or that,’” recalls Buick, who was hesitant to enter the social game at first. “That’s not what we wanted to build.”

Instagram was built to be a social network from the beginning and was thus inherently viral; this trumped any other features Hipstamatic may have had, including the originality of being the first to have millions of users taking filtered photos on the iPhone. This is a variation on a theme: [Startup = Growth](#). If you don’t consciously optimize your company for growth, you will be outgrown by a competitor that has done so. In particular, to maintain a constant monthly growth *rate*, you need to either keep hiring ever more salespeople of equal or greater quality (a very difficult task) or you need some way to grow virally, via your existing customer base. Virality thus means the ability to acquire more customers *without* a constantly expanding physical salesforce. It also means your economy of scale becomes vastly better, because you don’t need to budget for as much sales effort for each incremental customer.

The Virality Equation

In thinking about virality, there are three components to the virality equation:

- $p \in [0, 1]$: the probability that a given person decides to share
- $N \in [0, \infty)$: the number of people who the invite is shared with
- $\tau \in [0, \infty)$: the time interval between shares

Most people’s intuition tells them that the ideal way to build something highly viral is to improve the content, namely improving the value of p . However, suppose that we quantify the number of users at successive timepoints who have seen the content via $U(t)$ as follows, with $U(0) = 1$:

$$\begin{aligned} U(0) &= 1 \\ U(\tau) &= (Np) \\ U(2\tau) &= (Np)^2 \\ U(m\tau) &= (Np)^m \\ U(t) &= (Np)^{t/\tau} \end{aligned}$$

Here we have substituted $t = m\tau$ in the final line. We see immediately that if $K = Np < 1$ we do not have viral growth. We can also ask an important question: what’s the relative impact of increasing p by 2-fold, increasing N by 2-fold, or decreasing τ by 2-fold? For concreteness, let’s say that $p = .1$, $N = 50$, and $\tau = 1$ day. Then after three days we would have roughly:

$$U(3) = (.1 \times 50)^{3/1} = 125 \text{ users}$$

If we doubled p to .2 we'd get:

$$U(3) = (.2 \times 50)^{3/1} = 1000 \text{ users}$$

And if we doubled N to 100 we'd get:

$$U(3) = (.1 \times 100)^{3/1} = 1000 \text{ users}$$

But if we halved τ to .5 we'd get:

$$U(3) = (.1 \times 50)^{3/.5} = 15625 \text{ users}$$

Wow! The impact of reductions in τ is highly nonlinear because it affects the *exponent* in the virality equation. Shorter incubation times mean rapid viral spread. One can quantify this further by calculating partial derivatives of U with respect to N, p, τ but the point should be clear. A few further observations:

1. *Np must be greater than 1 to achieve viral growth.* It doesn't matter what your cycle time τ is if the underlying viral cycle doesn't spread to more than one person on average. Moreover, for a fixed $Np = K$, the larger the value of N and the smaller the value of p the more noisy and **stochastic** the viral spread is.
2. *Increasing N is easier than increasing p .* While p is upper bounded at 1.0, N is not. And while improving p requires improving the quality of the webpage, it's often easier to increase N by simply setting a default of "share all".
3. *Decreasing τ at first is easy.* Often to decrease τ one can get some easy wins by rearranging a user signup flow or the like. After that point diminishing returns kick in.
4. *Users need an incentive to share.* Communication applications like email, Facebook, Paypal, or Skype are inherently viral⁵, in that people need to make their contacts sign up in order to use the app. With Dropbox or Google Drive, we're dealing with optionally viral services, where users can use the app in a standalone fashion but collaboration on documents is quite helpful. Finally, with not-obviously-viral apps (like Mint.com), there's no immediate incentive for users to share private information (like their finances); Mint **famously** scaled up nonvirally.
5. *Lowest common denominator increases N .* The more broadly appealing your content, and the simpler it is to process conceptually, the easier it will be to sustain a large N , low p strategy.
6. *Financial incentives increase p .* The most obvious way to increase p is to surrender some of your profit margin per customer to achieve rapid viral growth. This strategy was highly successful for Paypal, but make sure to do the worst case math on this.

⁵This is one reason behind **Zawinski's law**: "Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can." Put another way, programs which can communicate with other programs are more likely to spread virally.

7. *Invited users should see exactly the same content.* In order to get a true viral loop, it's important to preserve the exact context on the page that provoked your initial user to share. If the users they invite don't see the exact same item that stimulated their contact to share, they'll need to hunt around on the page and/or chat with their contact to determine what was of interest. This will kill your viral loop.

Putting these things together, we can start to gain more insight as to why the Reddit image macro is the Ebola meme. An image meme is instantly processed ($\tau \rightarrow 0$) and understood by a wide audience ($N \rightarrow \infty$). It thus need only be shared by a small fraction of people to achieve incredible viral growth. Here are a few more historical examples of successful viral campaigns that illustrate various optimizations of p , N , and τ :

- *Hotmail and the email signature.* One of the [earliest](#) and most successful examples of viral marketing on the internet was Tim Draper's suggestion to include a link and marketing pitch at the end of every email sent from Sabeer Bhatia and Jack Smith's Hotmail. "Get your free email at Hotmail" resulted in [explosive viral growth](#). Every single person in a person's address book now contributed to N .
- *Youtube and Flash Video.* There were many video sharing sites in 2005. Why did Youtube in particular take off where others did not? One key [technology choice](#) was to use the lowest common denominator of Flash Video, rather than forcing the end user to install a new plugin such as Quicktime. This seemingly simple decision radically reduced τ and improved Youtube's viral loop beyond all competitors.
- *Facebook's use of Gmail contacts.* Before the OAuth standard was developed, Facebook [asked directly](#) for your Gmail password to help find friends. They then bulk downloaded contacts and used this to build their friend graph. Google eventually caught on and [closed](#) the barn door after the cows had escaped. A classic example of N optimization.
- *Zynga's incentive structure.* In the early days of Zynga, Mark Pincus had observed that people wanted to use real money in online games, but traditional gaming companies like Blizzard were dead set on stamping out such transactions. He cloned the popular Chinese app Happy Farm to create Farmville and created Zynga, which included invite mechanics as standard; either one could pay for a new item, or one could invite/spam one's friends. These [tactics](#) increased both p and N , as any active user was bound to share at some point and the rewards they got increased in proportion to the number of people they shared with.

The concept of social, then, is intimately linked with virality. It is not enough that your user merely discusses your app with their friends; it must be compelling enough for at least some of your user's friends to in turn share with *their* friends. This is why VCs mention social constantly: because a successful social strategy offers the possibility of rapid viral growth.

Local

Of the three, local is arguably the laggard in terms of market impact. The term can be interpreted in two ways: does it refer to the use of a GPS⁶ signal (aka a location-based app)

⁶As a side note, the use of GPS for such things is a bit astonishing. For many years, the GPS signal was selectively degraded by the US military. Civilian airlines needed to do things like GPS triangulation to provide

or does it involve outfitting local businesses with technology? The former describes apps like Uber and Exec, while the latter refers to commerce apps like Square, with the combination manifest in sites like Yelp and Foursquare. The companies in the local space are a bit less well known than their social counterparts, so let's run through some of the highlights.

- *Google Maps*. One of the unambiguous heavyweights in the local space by either definition, Google Maps is by some measures the [most popular](#) mobile app on Earth. It has fought off all comers for years and is a sophisticated, polished product. However, recently Google put a foot wrong with their Maps pricing snafu, giving [new energy](#) to the Open Street Maps (OSM) open source competitor and forcing big [price cuts](#).
- *Loopt*. Acquired for \$40M by Green Dot, Loopt was one of the first YCombinator companies. Sam Altman saw the importance of GPS before others and spent enormous effort getting carrier deals, only to find that the iPhone's built-in GPS obviated much of his work (see [here](#)). This is a relatively rare example of a startup being directly killed by a big company's competing product; Kiko vs. Google Calendar is [another](#).
- *Zocdoc*. One of the most successful local business companies. They solve one problem (online physician appointment booking) and they solve the whole problem. This seemingly trivial issue actually requires significant integration with the doctor's office, as their entire schedule is dictated by their calendar, and calendaring is [actually](#) a surprisingly challenging distributed systems problem.
- *Opentable*. Another successful local business company, and one of the first in the local space. Did much of the blocking and tackling necessary to get restaurant reservations and menus online, again requiring a significant sales process with often non-tech-savvy local restauratiers.
- *Foursquare*. A bit surprisingly, Foursquare looks like it'll be a [letdown](#) for the investors. This is not unprecedented: Napster, Friendster, Digg, and Second Life were all ramping at one point before missing a turn. For whatever reason, the seeming no-brainer-monetization of checkins as the new loyalty cards hasn't materialized. The check-in concept might get revisited in a few years with new technology, if and when mobile payments become mainstream.
- *Uber*. Uber uses GPS to locate the user and the cab s/he has just hailed, showing them each other's location and taking care of the billing and transaction. Now raising at a deserved multibillion valuation, Uber has reinvented local transportation in the metro areas where it's debuted. It faces significant competition from up and coming ridesharing cos (especially Lyft), but is arguably the single most successful location-based app to date.
- *Exec*. Allows you to book a worker in realtime for \$25/hour and see them running to your destination (and/or carrying out errands) via GPS. The latter feature seems superfluous but is surprisingly helpful in providing confidence that your tasks are being carried out in the right place at the right time.

signals suitable for automatically landing planes. But in a move similar to the repeal of the NSF AUP, in May 2000 the government [stopped degrading](#) the GPS signal. Who would have known that within the span of a decade the technology built for guiding nuclear missiles to their destination would be repurposed for guiding travellers to Dunkin Donuts?

- *Topquest*. Notable primarily for being perhaps the first genuinely monetizable use of checkins, Topquest incentivized sharing of locations for frequent flyer miles and was [acquired](#) in 2011.
- *Yelp*. Provides reviews and star ratings of many local businesses. Now a public company, Yelp has been successful in many respects, especially in terms of usage. Their [Monocle app](#) remains one of the first and most useful augmented reality products out there. While Yelp has had issues making a profit, the company is too valuable to simply go bust. At some point they will likely be acquired for the value of their data, perhaps by Apple.
- *Groupon*. The darling of 2010, Groupon has fallen quite a bit since that time. They didn't seem like competitors to Square at the beginning (one was a coupons company, one a payments terminal) but Andrew Mason's [mission statement](#) directing Groupon towards becoming an "Operating System for Local Commerce" clearly put both companies on a collision course. The problem, though, is that Groupon hired too many salespeople and didn't fundamentally have an engineering DNA. As such it has been outcompeted by Square.
- *Square*. This is the company that is probably going to win local commerce, with semi-stationary products like [Square Checkout](#) and mobile products like their original [Square Reader](#). Unlike Groupon, Square is run by computer scientists; it thus has a faster metabolism, higher intelligence, and greater creativity than a purely sales-driven squad. This is particularly important in a rapidly evolving business area like local payments.

Local Commerce, the Graveyard of Startups

The main difference between local (in the sense of local commerce) and the others is that it's a large market but extremely difficult to attack scalably. Even Benchmark's Bill Gurley, who is [bullish](#) on local, says:

The playbook requires a deep understanding of the industry, access to all the key content and its structure, a targeted and experienced sales structure, and a willingness to invest in a market that may seem "niche" to the broader service provider. You have to "be willing to get your hands dirty."

Why? Because you can't write a script to automate the process of selling to small businesses. Small businesses are generally the last to adopt new technology, outside of government agencies⁷. As a small business owner you are pinned down with a constant stream of demanding customers. The upside from this glowing technological [doodad](#) shown to you by some 20-something whippersnapper is unclear, but the downside is that your restaurant may come to a halt and/or you may need to retrain your entire staff around this one new item. To give an atmospheric sense of what it's like to run a small business, read [this article](#) and [this one](#):

The failure of a small cafe is not a question of competence. It is a sad given.
The logistics of a food establishment that seats between 20 and 25 people (which

⁷There are many pockets of high-tech excellence in the US Federal Government (e.g. [NCBI](#)), but in general government procurement processes and regulations mandate purchases of the safe "industry standard" rather than the risky "next big thing". This innate conservatism means that in government agencies, more than anywhere else, no one ever gets fired for buying IBM (or Microsoft, nowadays).

roughly corresponds to the definition of “cozy”) are such that the place will stay afloat - barely - as long as its owners spend all of their time on the job. There is a golden rule, long cherished by restaurateurs, for determining whether a business is viable. Rent should take up no more than 25 percent of your revenue, another 25 percent should go toward payroll, and 35 percent should go toward the product. The remaining 15 percent is what you take home. There’s an even more elegant version of that rule: Make your rent in four days to be profitable, a week to break even. If you haven’t hit the latter mark in a month, close.

This starts to make clear why local businesses are a tough sell:

1. *In-person sales.* In general, local business owners do not spend much time on the internet looking for new technologies. So Adwords campaigns and the like will be of limited effect. You will almost certainly need a separate physical sales visit to each locale, which gets expensive and time-consuming rapidly.
2. *Low margins.* Your customers aren’t rich. You’ll need to recognize that most local businesses aren’t making much money. So unless you are careful, you are now spending a lot of manual sales effort for relatively small and low-value conversions.
3. *High customer-service overhead.* Local businesses have no engineers to do integration or upgrades; everyone is busy serving customers and mopping floors. It is true that the more essential your technology is to their business, the more likely they are to buy; however, the more essential the technology the more you will have to support it. For example, if their new internet cash register goes down, their business grinds to a halt. Yet these cash registers are physical devices, distributed all around the country, and not easily updated or retrofitted as new technology comes out. So make sure that your engineers don’t push an update that can’t be undone without physically traveling to and resetting a device.
4. *Uncertain gains.* For many local businesses, the transition costs for introducing a new technology (e.g. the aforementioned risk of a cash register crash) often overwhelm the ostensible benefits.

Let’s do a simple calculation to drive the point home of why local sales is hard. Say that you hire a salesperson at \$50,000 per year to visit 10 stores per day over the course of 10 hours. Perhaps you have a 10% conversion rate, which could be high. Then at 5 days per week, the salesperson is acquiring 5 new customers per week, or 250 over the course of a year. Each of the 250 customers must produce at least \$200 in annual profit to pay the \$50,000 sales salary. If you are a payments startup, you might collect on the order of 2% per swipe. And $\$200/.02$ is \$10000. This means you need to sign up 250 companies and do a total of $\$250 \times \$10000 = \$2.5$ million in transactions through your system to pay for one sales rep. This doesn’t include taxes, overhead, benefits, software engineering, or anything like that. Please also note that local business operate on **tight margins**, and your 2% fee may need to be split with Visa or Mastercard, so even 2% is not assured. Finally, this presumes a somewhat superhuman salesperson who can visit 10 people per day every day for an entire year at only \$50,000 per year, with no commission.

This is why local sales is difficult.

There are ways to partially ameliorate these issues, such as local dinners where you can bulk-sell many proprietors at once, a subscription product, or the aggressive use of social networks to automate the process of gaining warm introductions to local business owners. But in general the problems associated with local sales compound. Due to low profit margins and the inability to hire skilled labor, the store manager is often the one responsible not just for running the business, but for promotion, marketing, the store website, and the like, including adoption of new technology.

One starts to understand why chains like Starbucks, with centralized marketing and significant division of labor, present such tough competition for local stores. As a startup, one also starts to understand why making the sale to a Starbucks, or even a much smaller chain, generally provides much more return-on-investment (ROI) than making the sale to a small business. One sale and you are (in theory) good to go at hundreds if not thousands of storefronts. And chains have some degree of internal engineering support, so they won't contact you for every breakage and issue. While it might be harder to get a chain to ink a large contract, once you do, you don't have to spend as much per-capita on customer-service or sales effort. This is why Square [did a deal](#) with Starbucks; even if the rollout has had problems ([1](#), [2](#)), solving those problems at scale will provide vastly more ROI than fixing the bugs of 1000 independent shops with all their idiosyncracies.

So, while it sounds like a great idea to sign up the enormous local business market, in practice these customers are the opposite of early adopters. They have thin margins, need high levels of customer service, and do not generally think of technology as a competitive advantage - especially when tech transition costs can dominate benefits. While you can certainly build something of significance in local business (Zocdoc, Zillow, Opentable, Grubhub, and the like are proof of this), you should only start a company in the area if you are comfortable with knocking on doors for the next five to ten years of your life.

Alternatively you can wait for the future of local to arrive. At some point within the next five years category-killers in each vertical will start turning local providers (doctors, drivers, restaurants) into APIs. Google will likely be heavily involved in the search and indexing of these APIs, but there promise to be many more interesting applications than even Google can contemplate.

To summarize, then: VCs tend to refer to two related but distinct things (GPS-based apps and local business apps) with the term *local*. The former is unambiguously technologically interesting and has led to large companies like Uber and fast-growing ones like Exec. The latter can in theory address large local business markets, but depends on laborious customer acquisition in specific verticals (like OpenTable, Zocdoc, Square) and can become a punishing sales grind.

Summary

Let's put it all together. Why do VCs love mobile, social, and local? Because a social app can have incredible viral growth, a mobile app is riding the worldwide smartphone/tablet phenomenon and has access to outstanding distribution, and (more arguably) a local app can use GPS in interesting ways and/or reach heretofore untouched small business markets. Some combination thereof can lead to rapid valuation growth in new markets, and hence substantial contribution to the VC's portfolio of hits. That is why these features are becoming *de rigueur*

for new companies. If you don't have a social and/or mobile component, it's like not having a website: your growth may be inherently limited relative to a competitor that does, even if their product is measurably inferior in other respects.

What will VCs love next? The obvious answer is [consumerization of the enterprise](#). And that's not a bad bet, and will involve a lot of blocking and tackling related to mobile device management ([MDM](#)), bring-your-own-device ([BYOD](#)) policies, [remote wipe](#), enterprise sales, and things of that nature. But what's more interesting are those things that involve fundamentally new technologies that flip tables and invalidate assumptions. In a subsequent lecture we will discuss the societal implications of several of these areas in some depth: industrial robotics, 3D printing, telepresence, quantified self, Bitcoin, and autonomous drones. These are our technologies of 2013.

Intermediate Javascript

In this lecture we'll go through the history of JS, talk about the important global objects and built-ins that are common across both client- and server-side JS implementations, and discuss the functional programming (FP) and object-oriented (OOP) paradigms in the context of JS.

The rise of Javascript

Javascript (JS) began as an add-on to Netscape, programmed in a few days by Brendan Eich in the early 1990s ([history](#), [more](#), [more](#)). Though JS is a completely different language from Java, Netscape used a similar syntax to Java and named it “Javascript” to piggy-back on Java’s marketing dollars:

Brendan Eich: The big debate inside Netscape therefore became “why two languages? why not just Java?” The answer was that two languages were required to serve the two mostly-disjoint audiences in the programming ziggurat who most deserved dedicated programming languages: the component authors, who wrote in C++ or (we hoped) Java; and the “scripters”, amateur or pro, who would write code directly embedded in HTML. Whether any existing language could be used, instead of inventing a new one, was also not something I decided. The diktat from upper engineering management was that the language must “look like Java”.

You can construct *some* rationale for this at the time ([Java applets](#) would be used for heavyweight web apps, with JS for lightweight page modifications), but in general the nomenclature issue mainly causes confusion. JS is not a stripped down version of Java, it is a completely different language.

And for much of its early history JS was not respected as a *serious* language. JS was seen as a way to annoy visitors with popup alerts and annoying animations. All that changed with the launch of Gmail in 2004. An influential [reverse-engineering](#) of Gmail and Google Maps by Jesse James Garrett in 2005 spawned the term AJAX, and it became clear that JS apps could actually accomplish some wondrous things. Even during the AJAX era, JS was still a client side language through the 2000s, with interpreters of the language mostly limited to the browser environment. In the mid-2000s, projects like [Mozilla Rhino](#) arose to provide a server-side JS environment, but it never caught on like [CPython](#) or [CRuby](#) (the most popular interpreters for the abstract Python and Ruby languages¹ respectively).

¹Though people often refer to them interchangeably, there is a difference between the Javascript programming language (an abstract entity specified by the ECMAScript [language specification](#)) and a particular Javascript interpreter (like Chrome’s embedded v8 interpreter, Mozilla Rhino, or node.js). Think of the language specification itself as giving a lowest common denominator that all interpreters must be able to parse and execute. On top of this, each vendor may then customize their interpreter to add JS extensions or new default libraries, above and beyond what is in the current specification. Often these extensions in turn become incorporated in the next specification. In this manner we have the cycle of innovation, then consensus, then innovation, and so on.

However, in the late 2000s a breakthrough in the optimization of dynamic languages called [trace trees](#) spurred the development of extremely fast JS compilers by both Google Chrome ([the v8 engine](#)) and Mozilla Firefox ([Tracemonkey](#)). Then in late 2009, an enterprising young engineer named Ryan Dahl took Chrome's open source v8 code, factored it out of the browser environment, and [announced](#) a new server-side JS environment on top of v8 called node.js. Not only did node popularize server-side programming in JS, it made native use of JS's facilities for non-blocking IO ([read more](#)), and is starting to become quite a [popular](#) choice among new startups - including the increasingly high-profile [Medium.com](#), by the founder of Blogger and co-founder of Twitter.

Basics and Built-ins

Let's now illustrate some intermediate JS concepts. Most of these examples should work both in a node environment and in a browser's JS prompt², with the exception of the `require` invocations. We assume you know how to program and have reviewed this [short tutorial](#) on MDN, where variable types, control flow primitives, conditional expressions, the syntax of function definitions, and so on are reviewed. Perhaps the best way³ to confirm that you understand these in the context of JS is to do a few problems from [Project Euler](#). Here's an example with [Euler problem 1](#):

```
1 #!/usr/bin/env node
2 // http://projecteuler.net/problem=1
3 // If we list all the natural numbers below 10 that are multiples of 3 or 5,
4 // we get 3, 5, 6 and 9. The sum of these multiples is 23.
5 //
6 // Find the sum of all the multiples of 3 or 5 below 1000.
7
8 var divides = function(a, b) {
9     return b % a === 0;
10}
11
12 var anydivide = function(as, b) {
13     for(var ii in as) {
14         if(divides(as[ii], b)) {
15             return true;
16         }
17     }
18     return false;
19}
20
21 var sum = function(arr) {
22     var cc = 0;
```

²In Chrome, you can view the JS prompt by going to the View Menu item, selecting Developer, and then selecting the Javascript Console. The variables in the webpage you're currently viewing will then be accessible at the prompt.

³If you want to do something for extra credit, it would also probably be quite useful to fork the [PLEAC](#) project and add Javascript. PLEAC predates node, and so many of the server-side examples can and should be redone in node now that there is a popular SSJS interpreter.

```

23     for(var ii in arr) {
24         cc += arr[ii];
25     }
26     return cc;
27 };
28
29 var fizzbuzz = function(factors, max) {
30     var out = [];
31     for(var nn = 1; nn < max; nn += 1) {
32         if(anydivide(factors, nn)) {
33             out.push(nn);
34         }
35     }
36     return sum(out);
37 };
38
39 console.log(fizzbuzz([3, 5], 1000));

```

And here's one with Euler problem 2:

```

1 #!/usr/bin/env node
2 // http://projecteuler.net/problem=2
3 //
4 // Each new term in the Fibonacci sequence is generated by adding the
5 // previous two terms. By starting with 1 and 2, the first 10 terms will be:
6 //
7 // 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
8 //
9 // By considering the terms in the Fibonacci sequence whose values do not
10 // exceed four million, find the sum of the even-valued terms.
11
12 // Fibonacci: closed form expression
13 // wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence
14 var fibonacci = function(n) {
15     var phi = (1 + Math.sqrt(5))/2;
16     return Math.round((Math.pow(phi, n + 1) - Math.pow(1 - phi, n + 1))/Math.sqrt(5));
17 };
18
19 var iseven = function(n) {
20     return n % 2 === 0;
21 };
22
23 var sum = function(arr) {
24     var cc = 0;
25     for(var ii in arr) {
26         cc += arr[ii];
27     }

```

```

28     return cc;
29 };
30
31 var fibsum = function(max) {
32     var value = 0;
33     var ii = 1;
34     var out = [];
35     var flag = false;
36     while(value < max) {
37         value = fibonacci(ii);
38         flag = iseven(value);
39         ii += 1;
40         if(flag && value < max) {
41             out.push(value);
42         }
43     }
44     return sum(out);
45 };
46
47 console.log(fibsum(4e6));

```

After a few of these [Project Euler](#) problems, you should feel comfortable with doing basic arithmetic, writing recursive functions, doing some string processing, and translating general CS algorithms into JS.

Array

Next, let's dive into some JS specifics, starting with the built-in [global objects](#) present in all full-fledged JS environments and specifically beginning with [Array](#). The [Array](#) global has some fairly self-evident methods that allow us to populate, shrink, expand, and concatenate lists of items.

```

1 // Important Array global methods
2
3 // length
4 var log = console.log;
5 var foo = [1, 2, 3, 3, 6, 2, 0];
6 log(foo.length); // 7
7
8 // concat: create new array
9 var bar = [4, 5, 6];
10 var baz = foo.concat(bar);
11
12 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
13 log(bar); // [ 4, 5, 6 ]
14 log(baz); // [ 1, 2, 3, 3, 6, 2, 0, 4, 5, 6 ]
15

```

```

16 // push/unshift to add to end/beginning
17 foo.push(10);
18 foo.unshift(99);
19 log(foo); // [ 99, 1, 2, 3, 3, 6, 2, 0, 10 ]
20
21 // pop/shift to remove the last/first element
22 var last = foo.pop();
23 var first = foo.shift();
24 log(last); // 10
25 log(first); // 99
26 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
27
28 // join: combine elements in an array into a string
29 log('<tr><td>' + bar.join('</td><td>') + "</td></tr>");
30 // <tr><td>4</td><td>5</td><td>6</td></tr>
31
32 // slice: pull out a subarray
33 var ref = foo.slice(3, 6);
34 log(ref); // [ 3, 6, 2 ]
35 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
36 ref[0] = 999;
37 log(ref); // [ 999, 6, 2 ]
38 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
39
40 // reverse: MODIFIES IN PLACE
41 foo.reverse();
42 log(foo); // [ 0, 2, 6, 3, 3, 2, 1 ]
43 foo.reverse();
44 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
45
46 // sort: MODIFIES IN PLACE
47 foo.sort();
48 log(foo); // [ 0, 1, 2, 2, 3, 3, 6 ]
49
50 // inhomogeneous types work
51 var arr = [99, 'mystr', {'asdf': 'alpha'}];
52
53 // JS will do the best it can if you invoke default methods.
54 // So make sure to use the REPL if you're doing something exotic.
55 arr.sort();
56 log(arr); // [ 99, { asdf: 'alpha' }, 'mystr' ]
57 arr.join(',') // '99,[object Object],mystr'
58
59
60 // Be very careful in particular when working with inhomogeneous arrays that
61 // contain objects. Methods like slice will not do deep copies, so modifications
62 // of the new array can propagate back to the old one when dealing with objects.

```

```

63 var newarr = arr.slice(0, 2);
64 console.log(newarr);      // [ 99, { asdf: 'alpha' } ]
65 console.log(arr);        // [ 99, { asdf: 'alpha' }, 'mystr' ]
66 newarr[1].asdf = 'ooga';
67 console.log(newarr);      // [ 99, { asdf: 'ooga' } ]
68 console.log(arr);        // [ 99, { asdf: 'ooga' }, 'mystr' ]

```

Importantly, as shown above, unlike an array in C, an Array in JS can be heterogenous in that it contain objects of different types. It's useful to use the REPL to verify the results if you're doing any type conversions with elements in arrays, making copies of arrays, or working with arrays that have nested objects in them.

Date

The `Date` object is useful for doing arithmetic with dates and times. While this might seem simple, you generally don't want to implement these kinds of routines yourself; date and time arithmetic involves tricky things like leap years and daylight savings time. A popular convention for dealing with this is to think of all times in terms of milliseconds relative⁴ to the [Unix Epoch](#), namely January 1 1970 at 00:00:00 GMT. Let's go through a few examples of working with Dates, building up to a concrete example of doing a JSON request and parsing string timestamps into Date instances.

```

1 // 1. Constructing Date instances.
2 //
3 //     Make sure to use "new Date", not "Date" when instantiating.
4 //
5 var year = 2000;
6 var month = 02;
7 var day = 29;
8 var hour = 12;
9 var minute = 30;
10 var second = 15;
11 var millisecond = 93;
12 var milliseconds_since_jan_1_1970 = 86400 * 10 * 1000;
13 var iso_timestamp = '2003-05-23T17:00:00Z';
14
15 var dt0 = Date(); // BAD - just a string, not a Date instance
16 var dt1 = new Date(); // GOOD - a real Date instance using new
17 var dt2 = new Date(milliseconds_since_jan_1_1970);
18 var dt3 = new Date(iso_timestamp);
19 var dt4 = new Date(year, month, day);
20 var dt5 = new Date(year, month, day, hour, minute, second);

```

⁴The problem with this convention is that after 2147483647 milliseconds, we arrive at January 19, 2038 3:14:07 GMT. If a 32-bit integer has been used for keeping track of the Unix time, it will overflow after this time and the system will crash (because $2^{31} = 2147483648$). The solution is to use a 64-bit integer for all time arithmetic going forward, but many legacy systems may not be updated in time. This is called the [Year 2038 problem](#), like the Year 2000 problem.

```

21 /*
22 > dt1
23 Mon Aug 05 2013 07:19:45 GMT-0700 (PDT)
24 > dt2
25 Sat Jan 10 1970 16:00:00 GMT-0800 (PST)
26 > dt3
27 Fri May 23 2003 10:00:00 GMT-0700 (PDT)
28 > dt4
29 Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
30 > dt5
31 Wed Mar 29 2000 12:30:15 GMT-0800 (PDT)
32 */
33
34 // 2. Date classmethods - these return milliseconds, not Date instances.
35
36 // now: gives number of milliseconds since Jan 1, 1970 00:00 UTC
37 var milliseconds_per_year = (86400 * 1000 * 365);
38 var years_from_epoch = function(ms) {
39     console.log(Math.floor(ms/milliseconds_per_year));
40 };
41 years_from_epoch(Date.now()); // 43
42
43 // parse: takes in ISO timestamp and returns milliseconds since epoch
44 years_from_epoch(Date.parse('2003-05-23T17:00:00Z')); // 33
45
46 // UTC: Constructor that returns milliseconds since epoch
47 years_from_epoch(Date.UTC(year, month, day, hour, minute,
48                             second, millisecond)); // 30
49
50
51 // 3. Date Examples
52
53 // 3.1 - Calculating the difference between two dates
54 console.log(dt3); // Fri May 23 2003 10:00:00 GMT-0700 (PDT)
55 console.log(dt4); // Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
56 var ddt = dt3 - dt4;
57 var ddt2 = dt3.getTime() - dt4.getTime();
58 console.log(ddt); // 99392400000
59 console.log(ddt2); // 99392400000
60 console.log(ddt / milliseconds_per_year); // 3.151712328767123
61
62 // 3.2 - Get JSON data, parsing strings into Date instances,
63 //        and then return docs structure.
64 var data2docs = function(data) {
65     var docs = [];
66     var offset = 'T12:00:00-05:00'; // Explicitly specify time/timezone.
67

```

```

68  var dt, ii, dtnew;
69  for(ii = 0; ii < data.results.length; ii += 1) {
70      dt = data.results[ii].publication_date; // looks like: '2012-12-14'
71      dti = new Date(dt + offset);
72      docs.push({'title':data.results[ii].title, 'publication_date': dti});
73  }
74  return docs;
75};

76
77 var docs2console = function(docs) {
78     for(var ii = 0; ii < docs.length; ii++) {
79         doc = docs[ii];
80         console.log('Date: %s\nTitle: %s\n', doc.publication_date, doc.title);
81     }
82};
83
84 var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
85 "03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
86 "&fields%5B%5D=publication_date";
87 var request = require('request'); // npm install request
88 var printdata = function(apiurl) {
89     request(apiurl, function (error, response, body) {
90         if (!error && response.statusCode == 200) {
91             data = JSON.parse(body);
92             docs2console(data2docs(data));
93         }
94     });
95 };
96
97 /* This produces the following output:
98 Date: Fri May 23 2003 10:00:00 GMT-0700 (PDT)
99 Title: National Security Agency/Central Security Service (NSA/CSS) Freedom of Information Act
100
101 Date: Fri Dec 14 2012 09:00:00 GMT-0800 (PST)
102 Title: Human Rights Day and Human Rights Week, 2012
103
104 Date: Mon Oct 20 2008 10:00:00 GMT-0700 (PDT)
105 Title: Tarp Capital Purchase Program
106 */

```

Note several things that we did here:

- In the last example, we added in an explicit time and timezone when converting the string timestamps into Date objects. This is for reproducibility; if you don't specify this then JS will use the local timezone.
- The number of milliseconds since the Unix Epoch is a bit hard to verify at first glance.

We defined some subroutines which use the fact that there are 86400 seconds per day to do some quick sanity checks.

- Differences between Dates are not Dates themselves, and you have to be careful about working with them. There are [modules](#) that can help with this, of which [moment.js](#) is the most popular.

You don't need to memorize all the Date methods; just know that you probably want to convert timestamps into Date instances before doing things with them, and also make heavy use of the REPL when working with Dates to confirm that your code is not automatically guessing times or timezones against your intention.

RegExp

The [RegExp](#) object is used to specify regular expressions. It uses Perl-inspired regular expressions (though [not](#) full PCRE) and can be used to recognize patterns in strings as well as to do basic parsing and extraction. Here are some examples⁵ of its use:

```
1 // Regexp examples
2
3 // 1. Test for presence, identify location
4 var ex1 = "The quick brown fox jumped over the lazy dogs.";
5 var re1 = /(cr|l)azy/;
6 console.log(re1.test(ex1));
7 // true
8 console.log(re1.exec(ex1));
9 /*
10 [ 'lazy',
11   'l',
12   index: 36,
13   input: 'The quick brown fox jumped over the lazy dogs.' ]
14 */
15
16
17 // 2. Global matches
18 var ex2 = "Alpha Beta Gamma Epsilon Omicron Theta Phi";
19 var re2 = /(\w+a )/;
20 var re2g = /(\w+a )/g;
21
22 re2.exec(ex2);
23 re2.exec(ex2);
24 /* Without the /g, repeating the match doesn't do anything here.
25
26 [ 'Alpha ',
27   'Alpha ',
28   index: 0,
```

⁵Note that in the very last example, we make use of some of the methods from the [underscore.js](#) library. Read the section on JS Functions for more on this.

```

29     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
30
31 [ 'Alpha ',
32   'Alpha ',
33   index: 0,
34   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
35 */
36
37 re2g.exec(ex2);
38 re2g.exec(ex2);
39 re2g.exec(ex2);
40 re2g.exec(ex2);
41 re2g.exec(ex2);
42 /* With the /g, repeating the match iterates through all matches.
43
44 [ 'Alpha ',
45   'Alpha ',
46   index: 0,
47   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
48
49 [ 'Beta ',
50   'Beta ',
51   index: 6,
52   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
53
54 [ 'Gamma ',
55   'Gamma ',
56   index: 11,
57   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
58
59 [ 'Theta ',
60   'Theta ',
61   index: 33,
62   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
63
64 null
65 */
66
67 // We can formalize the process of iterating through the
68 // matches till we hit a null with the following function:
69 var allmatches = function(rex, str) {
70   var matches = [];
71   var match;
72   while(true) {
73     match = rex.exec(str);
74     if(match !== null) { matches.push(match); }
75     else { break; }

```

```

76     }
77     return matches;
78 };
79 allmatches(re2g, ex2);
80 /*
81 [ [ 'Alpha ',
82   'Alpha ',
83   index: 0,
84   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
85 [ 'Beta ',
86   'Beta ',
87   index: 6,
88   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
89 [ 'Gamma ',
90   'Gamma ',
91   index: 11,
92   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
93 [ 'Theta ',
94   'Theta ',
95   index: 33,
96   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ] ]
97 */
98
99
100 // 3. Case-insensitive
101 var ex3 = "John is here.";
102 var re3s = /JOHN/;
103 var re3i = /JOHN/i;
104 re3s.test(ex3); // false
105 re3i.test(ex3); // true
106
107
108 // 4. Multiline
109 var ex4 = "Alpha beta gamma.\nAll the king's men.\nGermany France Italy.";
110 var re4s = /^G/;
111 var re4m = /^G/m;
112 re4s.test(ex4); // false
113 re4m.test(ex4); // true
114
115
116 // 5. Parsing postgres URLs
117 var pgurl = "postgres://myuser:mypass@example.com:5432/mydbpass";
118 var pgregex = /postgres:\//([^\:]+)([^\@]+)@([^\:]+)(\d+)\//(.+);
119 var flag = pgregex.test(pgurl); // true
120 var out = pgregex.exec(pgurl);
121 /*
122 [ 'postgres://myuser:mypass@example.com:5432/mydbpass',

```

```

123   'myuser',
124   'mypass',
125   'example.com',
126   '5432',
127   'mydbpass',
128   index: 0,
129   input: 'postgres://myuser:mypass@example.com:5432/mydbpass' ]
130 */
131
132
133 // 6. Parsing postgres URLs in a function (more advanced)
134 //     Here, we use the object and zip methods from underscore to organize
135 //     the regex-parsed fields in a nice, easy-to-use data structure.
136 var uu = require('underscore');
137 var parsedburl = function(dburl) {
138   var dbregex = /([^\:]+)\:\/\(([^\:]+)([^@]+)@([^\:]+)(\d+)\:([^\:]+)/;
139   var out = dbregex.exec(dburl);
140   var fields = ['protocol', 'user', 'pass', 'host', 'port', 'dbpass'];
141   return uu.object(uu.zip(fields, out.slice(1, out.length)));
142 };
143
144 console.log(parsedburl(pgurl));
145 /*
146 { protocol: 'postgres',
147   user: 'myuser',
148   pass: 'mypass',
149   host: 'example.com',
150   port: '5432',
151   dbpass: 'mydbpass' }
152 */

```

In general, it's more convenient to use the forward-slash syntax than to actually write out RegEx. Note that if you are heavily using regexes for parsing files, you may want to write a formal parser instead, use a csv or xml library, or make use of the built-in JSON parser.

Math

The `Math` object holds some methods and constants for basic precalculus. Perhaps the most useful are `floor`, `ceil`, `pow`, and `random`.

```

1 // Math examples
2
3 // 1. Enumerating the available Math methods
4 //     Object.getOwnPropertyNames allows you to do something like Python's dir.
5 var log = console.log;
6 log(Object.getOwnPropertyNames(Math));
7

```

```

8  /*
9   [ 'E',
10  'LN10',
11  'LN2',
12  'LOG2E',
13  'LOG10E',
14  'PI',
15  'SQRT1_2',
16  'SQRT2',
17  'random',
18  'abs',
19  'acos',
20  'asin',
21  'atan',
22  'ceil',
23  'cos',
24  'exp',
25  'floor',
26  'log',
27  'round',
28  'sin',
29  'sqrt',
30  'tan',
31  'atan2',
32  'pow',
33  'max',
34  'min' ]
35 */
36
37
38 // 2. Generating random integers between a and b, not including the upper bound.
39 var randint = function(a, b) {
40     var frac = Math.random();
41     return Math.floor((b-a)*frac + a);
42 };
43
44
45 // 3. Recapitulating constants
46 Math.pow(Math.E, Math.LN2); // 1.9999999999999998
47 Math.pow(Math.E, Math.LN10); // 10.000000000000002
48 Math.pow(10, Math.LOG10E) - Math.E; // 0
49 Math.pow(2, Math.LOG2E) - Math.E; // 0
50
51
52 // 4. Determining the effective growth rate from the start time (in months),
53 // the stop time (in months), the initial user base, and the final user
54 // base.

```

```

55 //  

56 // init * 2^{(stop-start)/tau} = fin  

57 //  

58 // tau = (stop-start)/log2(fin/init)  

59 //  

60 var log2 = function(xx) {  

61   return Math.log(xx)/Math.LN2;  

62 };  

63  

64 var doublingtime = function(start, stop, init, fin) {  

65   var dt = (stop-start);  

66   var fold = fin/init;  

67   return dt/log2(fold);  

68 };  

69  

70 log(doublingtime(0, 10, 1, 16)); // 2.5  

71 var tau = doublingtime(0, 24, 1, 9); // 7.571157042857489  

72 Math.pow(2, 24/tau); // 9.000000000000002

```

For the basics this is fine, but in practice you probably don't want to do too much math⁶ in JS. If you have heavily mathematical portions of your code, you might be able to implement them in Python via the `numpy` and `scipy` libraries (or the new `blaze`) and expose them over a `simple webservice` if you can tolerate the latency of an HTTP request. The other alternative is to implement your numerical code in C or C++ and then link it into JS via the built-in capability for C/C++ `addons` or a convenience library like the node foreign function interface (`node-ffi`).

String

The `String` object provides a few basic methods for string manipulation:

```

1 // String examples  

2  

3 // 1. Treat String as Array of characters  

4 var log = console.log;  

5 var sx1 = "The quick brown fox jumped over the lazy dogs.";  

6 sx1.charAt(10); // 'b'  

7 sx1.slice(10, 13); // 'bro'  

8 sx1.substring(10, 13); // 'bro'  

9 sx1.substr(10, 13); // 'brown fox jum'  

10 sx1.length; // 46

```

⁶If you are interested in rendering math with JS, look at the powerful `MathJax` library for generating \LaTeX in the browser. For server-side math, the new `mathjs` library might be worth checking out, as are the `matrix` libraries. However, math in node is very much in its infancy and numerical linear algebra is hard, subtle, and highly `architecture dependent`. So you probably want to rely on a heavily debugged external library like Python's `numpy` or the `GNU Scientific Library (GSL)` codebase, and then bridge it into node via a system-call, a webservice, the built-in provision for addons, or a foreign-function interface library (see text).

```

11 // 2. Compare strings using alphabetical ordering
12 var sx2 = "alpha";
13 var sx3 = "beta";
14 log(sx2 < sx3); // true
15
16 // 3. Replace substrings via string or regex
17 var sx4 = sx2.replace('ph', 'foo');
18 log(sx2); // 'alpha'
19 log(sx4); // 'alfooa'
20
21 var sx5 = sx2.replace(/a$/ , 'bar'); // NOTE regex
22 log(sx2); // 'alpha'
23 log(sx4); // 'alphbar'
24
25 // 4. Change case (lower, upper)
26 log(sx2.toUpperCase()); // 'ALPHA'
27
28 // 5. Trim strings
29 var sx6 = " " + ['Field1', 'Field2', 'Field3'].join("\t") + "\n";
30 var sx7 = sx6.trimRight();
31 var sx8 = sx6.trimLeft();
32 var sx9 = sx6.trim();
33 log(sx6); // 'Field1\tField2\tField3\n'
34 log(sx7); // 'Field1\tField2\tField3'
35 log(sx8); // 'Field1\tField2\tField3\n'
36 log(sx9); // 'Field1\tField2\tField3'
37
38 // 6. Split strings (useful for simple parsing)
39 var fields = sx9.split("\t");
40 log(fields); // [ 'Field1', 'Field2', 'Field3' ]

```

Again, for the basics this is fine, but for extremely heavy string manipulation you can lean on a library like the ones listed [here](#), particularly the popular [underscore-string](#).

JSON

The [JSON](#) global is used for rapidly parsing Javascript Object Notation (JSON), a subset⁷ of JS used for serializing data to disk and communicating between programming languages. JSON has replaced XML for most new applications because it's more human-readable than XML, easier to parse, doesn't require an (often-missing) separate [DTD](#) file to interpret the document, and has wide language support.

```

1 // JSON examples
2

```

⁷If you want to be a language lawyer, JSON is not [technically](#) a strict subset of JS. However, for most purposes it can be treated as such.

```

3 // 1. List methods on JSON object
4 var log = console.log;
5 Object.getOwnPropertyNames(JSON) // [ 'parse', 'stringify' ]
6
7 // 2. Using JSON.parse to deserialize JSON strings.
8 //
9 // Note that you use double quotes within JSON and single quotes to
10 // encapsulate the entire string
11 var jsdata = '[ {"asdf":9, "bar":10}, 18, "baz"]';
12 var data = JSON.parse(jsdata);
13 log(data[0].asdf); // 9
14
15 // You can also do this with eval. But don't do that. Use JSON.parse instead.
16 var data2 = eval(jsdata);
17 log(data2[0].asdf); // 9
18
19 // 3. Using JSON.stringify to serialize JS objects.
20 //
21 // While strings, numbers, arrays, and dictionaries/objects are generally
22 // safe, note that Regexp instances don't have a good default
23 // serialization and thus need special handling.
24 var dt = new Date('2003-05-23T17:00:00Z');
25 var rex = /(cr|l)/;
26 var data3 = [9, {"foo": dt, "bar": rex, "baz": {"quux": "a", "alpha": [77, 3]}}, 11];
27 log(data3);
28 /*
29 [ 9,
30   { foo: Fri May 23 2003 10:00:00 GMT-0700 (PDT),
31     bar: /(cr|l)/,
32     baz: { quux: 'a', alpha: [Object] } },
33   11 ]
34 */
35
36 // Note that the Regexp instance is serialize to {} rather than /(cr|l)/"
37 var data3str = JSON.stringify(data3);
38 // '[9,{"foo":"2003-05-23T17:00:00.000Z","bar":{},"baz":{"quux":"a","alpha":[77,3]}},11]'
39
40 // We can restore the data structure as shown. Note again the the restoration
41 // is only as good as the serialization. Make sure to look at the raw JSON string
42 // output
43 var data4 = JSON.parse(data3str);
44 log(data4);
45 /*
46 [ 9,
47   { foo: '2003-05-23T17:00:00.000Z',
48     bar: {},
49     baz: { quux: 'a', alpha: [Object] } },

```

```
50      11 ]
51 */
```

As a rule of thumb, you can consider using XML instead of JSON if you are actually marking up a document, like a novel or a newspaper article. But for other data interchange purposes you should usually use JSON.

Error

The `Error` object is used for exception handling; see in particular MDN's page on `throw` for some good examples related to invalid input types, along with the idea of rethrowing an exception.

```
// Error examples

// 1. List all methods in Error
Object.getOwnPropertyNames(Error);
/*
[ 'length',
  'name',
  'arguments',
  'caller',
  'prototype',
  'captureStackTrace',
  'stackTraceLimit' ]

// 2. An example of try/catch
var log = console.log;
var div = function(a, b) {
  try {
    if(b === 0) {
      throw new Error("Divided by Zero");
    } else {
      return a/b;
    }
  } catch(e) {
    log(`name\n${e.name}\nmessage\n${e.message}\nstack\n${e.stack}`);
  }
};

/*
name
Error
message
undefined
```

```

35
36 stack
37 Error: Divided by Zero
38     at div (repl:1:79)
39     at repl:1:2
40     at REPLServer.self.eval (repl.js:112:21)
41     at Interface.<anonymous> (repl.js:239:12)
42     at Interface.EventEmitter.emit (events.js:95:17)
43     at Interface._onLine (readline.js:202:10)
44     at Interface._line (readline.js:531:8)
45     at Interface._ttyWrite (readline.js:767:16)
46     at ReadStream.onkeypress (readline.js:99:10)
47     at ReadStream.EventEmitter.emit (events.js:98:17)
48 */
49
50
51 // 3. Returning an Error object directly
52 var div2 = function(a, b) {
53     if(b === 0) {
54         return new Error("Divided by Zero");
55     } else {
56         return a/b;
57     }
58 };
59 var err = div2(4, 0);
60 log(err.stack);
61 /*
62 Error: Divided by Zero
63     at div2 (repl:1:62)
64     at repl:1:11
65     at REPLServer.self.eval (repl.js:112:21)
66     at repl.js:249:20
67     at REPLServer.self.eval (repl.js:122:7)
68     at Interface.<anonymous> (repl.js:239:12)
69     at Interface.EventEmitter.emit (events.js:95:17)
70     at Interface._onLine (readline.js:202:10)
71     at Interface._line (readline.js:531:8)
72     at Interface._ttyWrite (readline.js:767:16)
73 */
74
75
76 // 4. Using custom error types.
77 //
78 // Modified from Zip Code example here:
79 // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw#Example
80 //
81 // Note that for a real email parser, you will want to use an existing

```

```

82 // regex rather than writing your own, as email formats can get
83 // surprisingly complex. See RFC 2822 and here:
84 //
85 // http://www.regular-expressions.info/email.html
86 // http://tools.ietf.org/html/rfc2822#section-3.4.1
87 //
88 // Better yet, if you really want to diagnose problems with invalid email
89 // addresses use a full function that returns specific errors for different
90 // cases. See here: stackoverflow.com/q/997078
91
92 var Email = function(emstr) {
93     var regex = /([^\@]+)\@([^\.]+\.)\.( [^\.]+)/;
94     if (regex.test(emstr)) {
95         var match = regex.exec(emstr);
96         this.user = match[1];
97         this.domain = match[2];
98         this.tld = match[3];
99         this.valueOf = function() {
100             return this.value;
101         };
102         this.toString = function() {
103             return this.user + '@' + this.domain + '.' + this.tld;
104         };
105     } else {
106         throw new EmailFormatException(emstr);
107     }
108 };
109
110 var EmailFormatException = function(value) {
111     this.value = value;
112     this.message = "not in a@b.c form";
113     this.toString = function() {
114         return this.value + this.message;
115     };
116 };
117
118 var EMAIL_INVALID = -1;
119 var EMAIL_UNKNOWN = -1;
120
121 var parseEmail = function(instr) {
122     try {
123         em = new Email(instr);
124     } catch (e) {
125         if (e instanceof EmailFormatException) {
126             return EMAIL_INVALID;
127         } else {
128             return EMAIL_UNKNOWN_ERROR;

```

```
129      }
130    }
131    return em;
132  };
133
134 // Make sure to include new when using the Email constructor directly
135 var foo = Email('john@gmail.com'); // Doesn't work
136 var bar = new Email('joe@gmail.com'); // Works
137
138 // Here's what happens when we invoke parseEmail
139 parseEmail('john@gmailcom'); // -1 == EMAIL_INVALID
140 parseEmail('johngmail.com'); // -1 == EMAIL_INVALID
141 parseEmail('john@gmail.com');
142 /*
143 { user: 'john',
144   domain: 'gmail',
145   tld: 'com',
146   valueOf: [Function],
147   toString: [Function] }
148 */
```

The try/catch paradigm is frequently used in browser JS, and you should be aware of it, but for node's callback heavy environment it's not ideal. We'll cover more on errors and exception in node specifically later. Be aware though that the `Error.stack` field is only available in IE10 and later (see [here](#) and [here](#)), and that you'll generally want to return instances of Error objects rather than using try/catch in node.

Built-in functions

In addition to these globals (Array, Date, RegExp, Math, String, JSON, Error), here are several miscellaneous built-in functions that come in for frequent use.

```
// Built-in examples

/*
 1. decodeURI and encodeURI: escape special characters in URIs

These two functions ensure that special characters like brackets and
slashes are escaped in URI strings. Let's illustrate with an
example API call to the federalregister.gov site.

*/
var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
"03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
"&fields%5B%5D=publication_date";
decodeURI(apiurl);
// 'https://www.federalregister.gov/api/v1/articles/03-12969,
// 2012-30312,E8-24781.json?fields[]=title&fields[]=publication_date'
```

```

16 encodeURI(decodeURI(apiurl));
17 // 'https://www.federalregister.gov/api/v1/articles/03-12969,
18 // 2012-30312,E8-24781.json?fields%5B%5D=title&fields%5B%5D=publication_date',
19 /*
20 2. typeof: gives the type of a JS variable.
21
22 Note: typeof cannot be used by itself, as it won't distinguish between
23 many kinds of objects (see here: http://stackoverflow.com/a/7086473). It
24 is, however, used by many libraries in conjunction with instanceof and
25 various kinds of duck-typing checks. See here:
26 tobyho.com/2011/01/28/checking-types-in-javascript
27
28 In general, metaprogramming in JS without a reliable library (something
29 like modernizr.js, but for types rather than browser features) is a pain
30 if you want to do it in full generality; it's not as regular as Python.
31 */
32
33
34 typeof 19
35 // 'number'
36 typeof "foo"
37 // 'string'
38 typeof {}
39 // 'object'
40
41 /*
42 3. parseInt and parseFloat: convert strings to ints and floats
43
44 Note that parseInt takes a base as the second argument, and can be used
45 to convert up to base 36.
46 */
47
48 parseInt('80', 10) // = 8*10^1 + 0*10^0
49 // 80
50 parseInt('80', 16) // = 8*16^1 + 0*16^0
51 // 128
52 parseFloat('89803.983')
53 // 89803.983
54
55 /*
56 4. Object.getOwnPropertyNames
57
58 Useful routine that can remind you which methods exist on a particular
59 object.
60 */
61
62 Object.getOwnPropertyNames(JSON)

```

```

63 // [ 'parse', 'stringify' ]
64
65
66 /*
67   5. Object.getOwnPropertyDescriptor
68
69   Use this to introspect the fields of a given object or module.
70 */
71
72 var uu = require('underscore');
73 Object.getOwnPropertyDescriptor(uu, 'map');
74 /*
75 { value: [Function],
76   writable: true,
77   enumerable: true,
78   configurable: true }
79 */

```

You can see a full list of [globals](#) and [builtins](#) at MDN; note in particular which ones are non-standard and which are available in all modern JS interpreters.

Functional Programming (FP) and JS Functions

Now that you have some familiarity with the built-ins, let's go through the various tricks and conventions related to JS functions, as well as several examples of doing functional programming in JS.

First-class functions

The concept of first-class functions (i.e. functions themselves as arguments and variables) is key to understanding modern JS. We've been using these for a while in different ways, but let's go through an example where we compare and contrast the functional programming (FP) style to the procedural style:

```

1 #!/usr/bin/env node
2
3 // Implementation 1: Procedural
4 var log = console.log;
5 var n = 5;
6 var out = [];
7 for(var i = 0; i < n; i++) {
8     out.push(i * i);
9 }
10 log(out); // [ 0, 1, 4, 9, 16 ]
11
12 // Implementation 2: Functional Programming
13 // Functions as first class variables

```

```

14 // 
15 // Allows us to abstract out the loop and the function applied within, such
16 // that we can swap in a new function or a new kind of iteration.
17 var sq = function(x) { return x * x;};
18 var cub = function(x) { return x * x * x;};
19 var loop = function(n, fn) {
20     var out = [];
21     for(var i = 0; i < n; i++) {
22         out.push(fn(i));
23     }
24     return out;
25 };
26 var loopeven = function(n, fn) {
27     var out = [];
28     for(var i = 0; i < n; i++) {
29         if(i % 2 === 0) {
30             out.push(fn(i));
31         } else {
32             out.push(i);
33         }
34     }
35     return out;
36 };
37 log(loop(n, sq));      // [ 0, 1, 4, 9, 16 ]
38 log(loop(n, cub));    // [ 0, 1, 8, 27, 64 ]
39 log(loopeven(n, sq)); // [ 0, 1, 4, 3, 16 ]
40 log(loopeven(n, cub)); // [ 0, 1, 8, 3, 64 ]
41
42
43 // Implementation 3: Functional Programming with underscorejs.org
44 // Note the use of the map and range methods.
45 // - range: generate an array of numbers between 0 and n
46 // - map: apply a function to a specified array
47 var uu = require('underscore');
48 log(uu.map(uu.range(0, n), sq)); // [ 0, 1, 4, 9, 16 ]
49 log(uu.map(uu.range(0, n), cub)); // [ 0, 1, 8, 27, 64 ]

```

By using so-called “higher order” functions (i.e. functions that accept other functions as arguments), we have cleanly decoupled the internal function and the external loop in this simple program. This is actually a very common use case. You may want to replace the loop with a graph traversal, an iteration over the rows of a matrix, or a recursive descent down the leaves of a binary tree. And you may want to apply an arbitrary function during that traversal. This is but one of many situations in which FP provides an elegant solution.

Functional programming and underscore.js

You can certainly do functional programming with the standard JS primitives, but it is greatly facilitated by the important [underscore.js](#) library. Let's redo the first Project Euler problem by making heavy use of underscore:

```
1 #!/usr/bin/env node
2 // http://projecteuler.net/problem=1
3 // If we list all the natural numbers below 10 that are multiples of 3 or 5,
4 // we get 3, 5, 6 and 9. The sum of these multiples is 23.
5 //
6 // Find the sum of all the multiples of 3 or 5 below 1000.
7 //
8 // We'll make heavy use of underscore here to illustrate some FP techniques.
9 var uu = require('underscore');
10
11 // For the anydivide function
12 // - uu.map takes an array and a univariate function, and applies the
13 //   function element-wise to the array, returning a new array.
14 //
15 // - We define divfn via a closure, so that it takes one variable and
16 //   returns one output. This can then be passed to uu.map.
17 //
18 // - uu.any takes an array of booleans and returns true iff one of them is
19 //   true, and false otherwise.
20 var anydivide = function(as, b) {
21     var divfn = function(a) { return b % a === 0; };
22     return uu.any(uu.map(as, divfn));
23 };
24
25 // For the sum function
26 //
27 // - uu.reduce takes an array and a function, and applies that function
28 //   iteratively to each element of the array, starting from the first element.
29 // - The anonymous function we pass in as the second argument just adds adjacent
30 //   elements together.
31 //
32 var sum = function(arr) {
33     return uu.reduce(arr, function(a, b) { return a + b; });
34 };
35
36 // For the fizzbuzz function
37 //
38 // - We define divfn as a closure, using the factors variable. Again, this
39 //   is so that it takes one argument, returns one output, and can be passed to map.
40 //
41 // - We use uu.filter to iterate over an array and return all elements
42 //   which pass a condition, specified by a function that returns booleans.
```

```

43 //      The array is the first argument to uu.filter and the boolean-valued
44 //      func is the second argument.
45 //
46 // - We use uu.range to generate an array of numbers between two specified
47 // - values.
48 //
49 var fizzbuzz = function(factors, max) {
50     var divfn = function(nn) { return anydivide(factors, nn); };
51     var divisible = uu.filter(uu.range(1, max), divfn);
52     return sum(divisible);
53 };
54
55 // console.log(fizzbuzz([3, 5], 10));
56 console.log(fizzbuzz([3, 5], 1000));

```

We've commented this much more heavily than we normally would. You can explore the full panoply of underscore methods [here](#). While the functional style might be a bit hard to understand at first, once you get the hang of it you'll find yourself using it whenever iterating over arrays, applying functions to arrays, generating new arrays, and the like. It makes your code more compact and the systematic use of first-class functions greatly increases modularity. In particular, by using the FP paradigm you can often⁸ reduce the entire internals of a program to the application of a single function to an external data source, such as a text file or a JSON stream.

Partial functions and the concept of currying

A common paradigm in functional programming is to start with a very general function, and then set several of the input arguments to default values to simplify it for first-time users or common situations. We can accomplish this via currying, as shown:

```

1 // 1. Partial function application in JS
2 //
3 // The concept of partial function application (aka currying, named after
4 // Haskell Curry the mathematician).
5 var sq = function(x) { return x * x; };
6 var loop = function(n, fn) {
7     var out = [];
8     for(var i = 0; i < n; i++) {
9         out.push(fn(i));
10    }
11    return out;
12 };
13

```

⁸Note that the asynchronous nature of node might seem to interfere with the idea of expressing the whole thing as `f(g(h(x)))`, as it seems instead that you'd have to express it as `h(x,g(y,f))` via callbacks. However, as we will see, with the appropriate flow control libraries and especially the pending introduction of the `yield` keyword in node, it again becomes feasible to code in a functional style.

```

14 // We can use the 'bind' method on Functions to do partial evaluation
15 // of functions, as shown.
16 //
17 // Here, loopN now accepts only one argument rather than two. (We'll cover
18 // the null argument passed in to bind when we discuss 'this').
19 loop(10, sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
20 var loopN = loop.bind(null, 10); // n = 10
21 loopN(sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
22
23 // 2. Implementing currying in underscore is easy with the 'partial' method.
24 var uu = require('underscore');
25 loop6 = uu.partial(loop, 6);
26 loop6(sq); // [ 0, 1, 4, 9, 16, 25 ]

```

Note that underscore provides tools for doing this as well. In general, if you want to do something in JS in the FP style, underscore is a very powerful tool.

Function scope

The scope where variables are defined is a little tricky in JS. By default everything is global, which means that you can easily have a namespace collision between two different libraries. There are many gotchas in this area, but you can generally get away with considering the following four ways of thinking about scope.

- global scope
- statically defined functions with statically defined scope
- dynamically defined functions with statically defined scope (closures)
- dynamically defined functions with dynamically defined scope (`this`)

Here, we use the term “statically defined” to indicate that the function or scope can be fully specified simply by inspecting the `.js` file (“static analysis”), whereas “dynamically defined” means that the function or scope can take on different values in response to user input, external files, random number generation, or other external variables that are not present in the source code `.js` file. Let’s see some examples of each of these:

```

1 // Scope in JS
2
3 // 1. Unintentional global variables
4 //
5 // This can occur if you forget to declare cc within foo.
6 var cc = 99;
7 var foo = function(aa) {
8     var bb = 7;
9     console.log("aa = " + aa);
10    console.log("bb = " + bb);

```

```

11     console.log("cc = " + cc);
12 };
13 foo(22);
14 /*
15 aa = 22
16 bb = 7
17 cc = 99
18 */
19
20 // 2. Intentional global variables
21 //
22 // This is the recommended syntax within node if you really
23 // want to use globals. You can accomplish the same thing with
24 // window in the browser.
25 var dd = 33;
26 var bar = function(aa) {
27     var bb = 7;
28     console.log("aa = " + aa);
29     console.log("bb = " + bb);
30     console.log("dd = " + global.dd);
31 };
32 bar(22);
33 /*
34 aa = 22
35 bb = 7
36 dd = 33
37 */
38
39
40 // 3. Scope in statically defined functions
41 //
42 // The aa defined within this functions' argument and the bb within its
43 // body override the external aa and bb. This function is defined
44 // 'statically', i.e. it is not generated on the fly from input arguments
45 // while the program is running.
46 var aa = 7;
47 var bb = 8;
48 var baz = function(aa) {
49     var bb = 9;
50     console.log("aa = " + aa);
51     console.log("bb = " + bb);
52 };
53 baz(16);
54 /*
55 aa = 16
56 bb = 9
57 */

```

```

58
59
60 // 4. Scope in dynamically defined functions (simple)
61 //
62 // Often we'll want to build a function on the fly, in which case the use
63 // of variables from the enclosing scope is a feature rather than a
64 // bug. This kind of function is called a closure. Let's first do a
65 // trivial example.
66 //
67 // Here, buildfn is intentionally using the increment variable
68 // passed in as an argument in the body of the newfn, which is
69 // being created on the fly.
70 var buildfn = function(increment) {
71     var newfn = function adder(xx) { return xx + increment; };
72     return newfn;
73 };
74 buildfn(10); // [Function: adder]
75 buildfn(10)(17); // 27
76 var add3 = buildfn(3);
77 var add5 = buildfn(5);
78 add3(add5(17)); // 25
79
80
81 // Here's a more complex example where we pass in a function rather
82 // than simply assuming buildfn2 will be doing addition.
83 var buildfn2 = function(yy, binaryoperator) {
84     var cc = 10;
85     var newfn = function binop(xx) { return cc*binaryoperator(xx, yy); };
86     return newfn;
87 };
88 var pow = function(aa, bb) { return Math.pow(aa, bb); }
89 var fn = buildfn2(3, pow);
90 fn(5); // 10 * Math.pow(3, 5)
91
92
93 // 5. Scope in dynamically defined functions (more advanced)
94 //
95 // Here's a more realistic example, where we define a meta-function that
96 // takes two other functions and snaps them together to achieve an
97 // effect.
98 var compose = function(f, g) {
99     var h = function(x) {
100         return f(g(x));
101     };
102     return h;
103 };
104

```

```

105 var jsdata = '[ {"asdf":9, "bar":10}, 18, "baz"]';
106 var f1 = function(data) { return data[0].bar + 11;};
107 var f2 = JSON.parse;
108 f1(f2(jsdata)); // 10 + 11 === 21
109 var f1f2 = compose(f1, f2);
110 f1f2(jsdata); // 10 + 11 === 21
111
112
113 // 6. Dynamically defined scope in functions (this)
114 //
115 // Now, what if we want to go one level more abstract? This time
116 // we want to define a closure that doesn't take on its values
117 // from the current context, but from some context that we will
118 // dynamically specify at a later date.
119 var bb = 10;
120 var static_closure = function(aa) {
121     return aa + bb;
122 };
123 static_closure(3);
124 // 13
125
126 var dynamic_closure = function(aa) {
127     return aa + this.bb;
128 };
129
130 var context1 = {
131     'fn': dynamic_closure,
132     'bb': 10
133 };
134
135 context1.fn(3);
136 // 13
137
138 var context2 = {
139     'fn': dynamic_closure,
140     'bb': Math.random()
141 };
142
143 context2.fn(3);
144 // 3.656272369204089 [your value will be different]
145
146
147 //
148 // Great. So, to recap, the 'this' variable points to a single object or
149 // else is undefined. You can use it to leave the enclosing scope for a
150 // closure unspecified until the last moment when it's executed.
151 //

```

```

152 // If you can, you may want to simply pass in the object itself as an
153 // explicit variable, as shown below. Otherwise one can use 'this'.
154 var simpler_than_dynamic_closure = function(aa, obj) {
155     return aa + obj.bb;
156 };
157
158 simpler_than_dynamic_closure(3, context2);
159 // 3.656272369204089

```

This example shows:

- how one can inadvertently use globals
- how to use globals intentionally if you really need to
- how scope works within standard statically defined functions
- how you can dynamically define functions via closures.
- how you can even dynamically define enclosing scope via the `this` keyword

While there are definitely times you do want to build closures, you don't want to inadvertently use global variables locally (or, more generally, have a scoping bug). There are three ways to ensure that this doesn't happen. First, invoke JSHint as described in Lecture 4b to confirm that all your variables have `var` in front of them; you can set up a `~/.jshintrc` to assist with this (see [here](#)). Second, use the node and Chrome JS REPLs aggressively to debug sections of code where you're unsure about the scope; you may need to do this quite a bit to understand the `this` keyword at the beginning. Third, you can and should aggressively encapsulate your code.

Encapsulation: the self-executing function trick

A very common idiom you will see in JS code is the self-executing function trick. We define a function and then immediately execute it, returning a single object that contains the results of that function:

```

1 // The concept of self-executing functions
2
3 // 1. Basic self-executing functions for encapsulation
4 //
5 // Suppose you have some JS code like this:
6 var foo = "inner";
7 console.log(foo);
8
9 // You worry that the external world may also have defined a foo variable.
10 // To protect your code, you can encapsulate it by putting it in a
11 // self-executing function.
12 var foo = "outer";
13 (function() {

```

```

14     var foo = "inner";
15     console.log(foo);
16 })();
17 console.log(foo);
18
19 // This will print:
20 /*
21 inner
22 outer
23 */
24
25 // Let's break this down bit by bit. The inside is just an anonymous
26 // function definition. Let's assign it to a variable.
27 var anon = function () {
28     var foo = "inner";
29     console.log(foo);
30 };
31
32 // We enclose this function definition in parentheses, and then immediately
33 // execute it by adding another two parens at the end. We also need a
34 // semicolon as the expression is ending.
35 (function() {
36     var foo = "inner";
37     console.log(foo);
38 })();
39
40 // 2. Passing in variables to a self-executing function
41 //
42 // Now that we've set up a wall, we can do something more sophisticated by
43 // passing in specific variables from the enclosing scope.
44 var bar = "allowed";
45 (function(aa) {
46     var foo = "inner";
47     console.log(foo + " " + aa);
48 })(bar);
49
50 // This prints
51 /*
52 inner allowed
53 */
54
55 // 3. Passing in and receiving variables from a self-executing function.
56 //
57 // Now we'll return an object as well, converting the inside into a proper
58 // function.
59 var bar = "allowed";
60 var result = (function(aa) {

```

```

61  var foo = "inner";
62  var out = foo + " " + aa;
63  console.log(out);
64  return {"val": out};
65 })(bar);
66 console.log(result);
67
68 // This prints
69 /*
70 { val: 'inner allowed' }
71 */

```

This is a hack to compensate for the fact that JS, unlike (say) Python or Ruby, did not have a builtin module system. Today the combination of the node `require` syntax and npm gives a good solution for server-side modules, while browserify is a reasonable way of porting these into client-side modules (though development of modules within the ECMAScript standard continues apace).

Callbacks, asynchronous functions, and flow control

Now that you have a deeper understanding of the concept of first-class functions and functional programming, we can explain the concept of callbacks and then asynchronous functions. A callback is a function that is passed in as an argument to another function. When the second function completes, it feeds the results that would normally be returned directly into the arguments of the callback. Let's look at an example:

```

1  #!/usr/bin/env node
2
3  var request = require('request'); // npm install request
4  var apiurl = 'http://nodejs.org/api/index.json';
5  var callbackfn = function(error, response, body) {
6    if (!error && response.statusCode == 200) {
7      console.log(body);
8    }
9  };
10 request(apiurl, callbackfn);
11
12 // When executed, this produces the following
13 /*
14 {
15   "source": "doc/api/index.markdown",
16   "desc": [
17     {
18       "type": "list_start",
19       "ordered": false
20     },
21     {

```

```

22     "type": "list_item_start"
23 },
24 {
25     "type": "text",
26     "text": "[About these Docs](documentation.html)"
27 },
28 {
29     "type": "list_item_end"
30 },
31 {
32     "type": "list_item_start"
33 },
34 {
35     "type": "text",
36     "text": "[Synopsis](synopsis.html)"
37 },
38 {
39     "type": "list_item_end"
40 },
41 {
42     "type": "list_item_start"
43 },
44 ...
45 */

```

We may want to pass callbacks into callbacks. For example, suppose that we want to write the results of this API call to disk. Below is the messy way to do this. This works, but it is bad in a few respects. First, it's difficult to read. Second, it mixes three different functions together (making the HTTP request, extracting the body from the HTTP response, and writing the body to disk). Third, it is hard to debug and test in isolation. The messy entanglement below is what is known as “callback hell”.

```

1 #!/usr/bin/env node
2
3 var request = require('request');
4 var apiurl = 'http://nodejs.org/api/index.json';
5 var fs = require('fs');
6 var outfile = 'index2.json';
7 request(apiurl, function(error, response, body) {
8     if (!error && response.statusCode == 200) {
9         fs.writeFile(outfile, body, function (err) {
10             if (err) throw err;
11             var timestamp = new Date();
12             console.log("Wrote %s %s", outfile, timestamp);
13         });
14     }

```

```
15 } );
```

Now, there is one advantage of the callback approach, which is that by default it supports asynchronous execution. In this version we have wrapped the request in `setTimeout` just to show that even though the request is executed first, it runs in the background asynchronously. Code that is executed after the request can complete before it. This is the essence of asynchronous execution: a line of code does not wait for preceding lines of code to finish unless you explicitly force it to be synchronous.

```
1 #!/usr/bin/env node
2
3 var request = require('request');
4 var apiurl = 'http://nodejs.org/api/index.json';
5 var fs = require('fs');
6 var outfile = 'index3.json';
7 setTimeout(function() {
8     request(apiurl, function(error, response, body) {
9         if (!error && response.statusCode == 200) {
10             fs.writeFile(outfile, body, function (err) {
11                 if (err) throw err;
12                 var timestamp = new Date();
13                 console.log("Invoked first, happens second at %s", new Date());
14                 console.log("Wrote %s %s", outfile, timestamp);
15             });
16         }
17     });
18 }, 3000);
19 console.log("Invoked second, happens first at %s", new Date());
```

We can make this less messy as follows. It's still not as pretty as synchronous code, but will do for now.

```
1 #!/usr/bin/env node
2
3 var request = require('request');
4 var apiurl = 'http://nodejs.org/api/index.json';
5 var fs = require('fs');
6 var outfile = 'index4.json';
7
8 var cb_writefile = function (err) {
9     if (err) throw err;
10     var timestamp = new Date();
11     console.log("Invoked first, happens second at %s", new Date());
12     console.log("Wrote %s %s", outfile, timestamp);
13 };
14
```

```

15 var cb_parsebody = function(error, response, body) {
16   if (!error && response.statusCode == 200) {
17     fs.writeFile(outfile, body, cb_writefile);
18   }
19 };
20
21 request(apiurl, cb_parsebody);
22 console.log("Invoked second, happens first at %s", new Date());

```

Now suppose that we want to nicely format the results of this API call. Below is one reasonably good way to do this.

```

1 #!/usr/bin/env node
2
3 var uu = require('underscore');
4 var request = require('request');
5 var apiurl = 'http://nodejs.org/api/index.json';
6 var fs = require('fs');
7 var outfile = 'index-parsed.json';
8
9 var data2name_files = function(data) {
10   var module2obj = function(mod) {
11     var modregex = /\[(\[^\\]+)\]\((\[^\\]+)\)\/;
12     var match = modregex.exec(mod);
13     return {'name': match[1], 'file': match[2]};
14   };
15   var notUndefined = function(xx) { return !uu.isUndefined(xx); };
16   var modules = uu.filter(uu.pluck(data.desc, 'text'), notUndefined);
17   return uu.map(modules, module2obj);
18 };
19
20 var body2json = function(body) {
21   return JSON.stringify(data2name_files(JSON.parse(body)), null, 2);
22 };
23
24 var cb_writefile = function (err) {
25   if (err) throw err;
26   var timestamp = new Date();
27   console.log("Invoked first, happens second at %s", new Date());
28   console.log("Wrote %s %s", outfile, timestamp);
29 };
30
31 var cb_parsebody = function(error, response, body) {
32   if (!error && response.statusCode == 200) {
33     fs.writeFile(outfile, body2json(body), cb_writefile);
34   }
35 };

```

```

36
37 request(apiurl, cb_parsebody);
38 console.log("Invoked second, happens first at %s", new Date());
39
40
41 /* This code turns the raw output:
42
43     {
44         "source": "doc/api/index.markdown",
45         "desc": [
46             {
47                 "type": "list_start",
48                 "ordered": false
49             },
50             {
51                 "type": "list_item_start"
52             },
53             {
54                 "type": "text",
55                 "text": "[About these Docs](documentation.html)"
56
57             ...
58
59     Into something like this:
60
61     [
62         {
63             "name": "About these Docs",
64             "file": "documentation.html"
65         },
66         {
67             "name": "Synopsis",
68             "file": "synopsis.html"
69         },
70         ...
71 ]*/

```

Note that to debug, we can write the raw JSON data pulled down by the first HTTP request to disk, and then load directly via `var data = require('./index.json');` or a similar invocation. We can then play with this data in the REPL to build up a function like `data2name_files`. One of the issues with the callback style is that it makes working in the REPL more inconvenient, so tricks like this are useful.

Again, the main advantage of the asynchronous paradigm is that it allows us to begin the next part of the program without waiting for the current one to complete, which [can improve performance](#) and responsiveness for certain classes of problems. The main disadvantages are that async programming alone [will not](#) improve performance if we are compute- rather than IO-bound, asynchrony can make programs [harder](#) to reason about, and asynchronous code

can lead to the ugly [nested callback](#) issue. One way around this is to use one of the node flow control libraries, like caolan's [async](#). Another is to wait for the [long-awaited](#) introduction of the `yield` keyword⁹ in node 0.11. To first order, once node 0.11 has a stable release, one should be able to replace `return` with `yield` and get the same conceptual and syntactical cleanliness of synchronous code, while preserving many of the useful features of asynchronous code.

Object-Oriented Programming (OOP), Prototypal Inheritance, and JS Objects

Now let's go through some of the basics of JS objects and the OOP style in JS. MDN as usual has a [good reference](#), and the [Eloquent JS](#) material is good as well; we'll complement and extend the definitions given therein.

Objects as dictionaries

The simplest way to work with objects is to think of them as similar to Python's dictionary type. We can use them to organize data that belongs together. Here's an example of a function that parses a URL scheme into its component pieces:

```

1  #!/usr/bin/env node
2  var parsedburl = function(dburl) {
3      var dbregex = /([:]*)+:(\/\/([:]*)+)([^@]+@[^:]+):(\d+)/(.+);
4      var out = dbregex.exec(dburl);
5      return {'protocol': out[1],
6              'user': out[2],
7              'pass': out[3],
8              'host': out[4],
9              'port': out[5],
10             'dbpass': out[6]};
11 };
12 var pgurl = "postgres://myuser:mypass@example.com:5432/mydbpass";
13 console.log(parsedburl(pgurl));
14 /*
15 { protocol: 'postgres',
16   user: 'myuser',
17   pass: 'mypass',
18   host: 'example.com',
19   port: '5432',
20   dbpass: 'mydbpass' }
21 */

```

⁹You can see this [post](#) and [this one](#). You'll want to invoke the sample code with `node --harmony` and also note that `gen.send()` may not yet be implemented. You can use `nvm install v0.11.5` to get the latest version of node and `nvm use v0.11.5` to switch to it for testing purposes, and then revert back with `nvm use v0.10.15`.

You can take a look at Tim Caswell's series on JS Object Graphs ([1](#), [2](#), [3](#)) if you want to get into the guts of exactly what's going on when objects are instantiated in this way, and specifically how references between and within objects work.

Objects as dictionaries with functions

Without adding too much in the way of new concepts, we can use the special `this` keyword we described above to add a bit of intelligence to our objects. Now they aren't just dumb dictionaries anymore.

```
1 var dbobj = {  
2   protocol: 'postgres',  
3   user: 'myuser',  
4   pass: 'mypass',  
5   host: 'example.com',  
6   port: '5432',  
7   dbpass: 'mydbpass',  
8   toString: function() {  
9     return this.protocol + '://' +  
10    this.user + ':' +  
11    this.pass + '@' +  
12    this.host + ':' +  
13    this.port + '/' +  
14    this.dbpass;  
15  }  
16};  
17  
18 dbobj.toString();  
19 // 'postgres://myuser:mypass@example.com:5432/mydbpass'
```

Here we equip the object with a function that pretty prints itself.

Objects as “classes”

Things start to get more complex when we start thinking about classes and inheritance. Unlike Java/C++/Python, JS is not a strictly object-oriented language. It actually has a prototypal inheritance model; see [here](#) for the difference. In practice, though, you can use it in much the same way that you'd create classes and class hierarchies in (say) Python.

The tricky part is that JS is flexible enough that you will see many different ways of setting up inheritance. Most of the time you'll want to either use simple objects-as-dictionaries (as in the previous section), or else use so-called [parasitic combination inheritance](#). Let's do an example with Items for an e-commerce site like Amazon, building on some of the convenience functions defined by [Crockford](#), [Zakas](#), and [Bovell](#) (specifically the somewhat magical and now-standardized `Object.create` and the related `inheritPrototype`).

In our Amazon-ish example, each `Item` has a [Stock Keeping Unit](#) number (SKU) and a price, but other than that can vary widely and support different kinds of functions. A `Book` instance, for example, can search itself while a `Furniture` instance can calculate the floor area

in square feet that they will occupy. Here's how we'd implement that; remember to use the `new` keyword when instantiating instances ([1](#), [2](#), [3](#)).

```
1 // Example of inheritance
2 //
3 // Item - sku, price
4 // Book - title, text, search
5 // Furniture - name, width, length
6
7
8 // 0. Preliminaries: this helper function copies the properties and methods
9 // of the parentObject to the childObject using the copyOfParent
10 // intermediary.
11 //
12 // See Zakas' writeup at goo.gl/o1wRGO for details.
13 function inheritPrototype(childObject, parentObject) {
14     var copyOfParent = Object.create(parentObject.prototype);
15     copyOfParent.constructor = childObject;
16     childObject.prototype = copyOfParent;
17 }
18
19 // 1. Define the parent class constructor and add prototype methods
20 function Item(sku, price) {
21     this.sku = sku;
22     this.price = price;
23 }
24 Item.prototype.toString = function () {
25     return "SKU: " + this.sku + " | Price: " + this.price + " USD";
26 };
27
28 // 2. Define the subclass constructor, copy over properties and methods of
29 // Item, and then define a new function.
30 function Book(sku, price, title, text) {
31     Item.call(this, sku, price);
32     this.title = title;
33     this.text = text;
34 };
35
36 inheritPrototype(Book, Item);
37
38 Book.prototype.search = function(regexstr) {
39     var regex = RegExp(regexstr);
40     var match = regex.exec(this.text);
41     var out = '';
42     if(match !== null) {
43         var start = match.index;
44         var end = match.index + match[0].length;
```

```

45     var dx = 3;
46     var padstart = start - dx > 0 ? start - dx : start;
47     var padend = end + dx > 0 ? end + dx : end;
48     out = '...' + this.text.slice(padstart, padend) + '...';
49   }
50   return out;
51 };
52
53 // 3. Do one more subclass for illustrative purposes
54 function Furniture(sku, price, name, width, length) {
55   Item.call(this, sku, price);
56   this.name = name;
57   this.width = width;
58   this.length = length;
59 };
60
61 inheritPrototype(Furniture, Item);
62
63 Furniture.prototype.floorArea = function() {
64   return this.width * this.length;
65 };
66
67
68 // 4. Now let's test things out.
69 // Remember to use new!
70 var foo = new Item("ID:8908308", 43.27);
71 foo.sku;           // 'ID:8908308'
72 foo.price;        // 43.27
73 foo.toString();   // 'SKU: ID:8908308 / Price: 43.27 USD'
74
75 var bible = new Book("ID:123456", 101.02, "The Bible", "In the beginning there was");
76 bible.sku;         // 'ID:123456'
77 bible.price;       // 101.02
78 bible.toString();  // 'SKU: ID:123456 / Price: 101.02 USD'
79 bible.search('there'); // '...ng there wa...'
80 bible.search('therex'); // ''
81
82 var chair = new Furniture("ID:79808", 2020.32, "A chair", .5, 4.2);
83 chair.sku;         // 'ID:79808'
84 chair.price;       // 2020.32
85 chair.toString();  // 'SKU: ID:79808 / Price: 2020.32 USD'
86 chair.floorArea(); // 2.1

```

This is a common way to do inheritance in JS. However, the syntax here is a bit kludgy and redundant, and a bit difficult to remember because JS doesn't have one true way of doing inheritance; see also the discussion in [Eloquent JS](#). You might want to build a small library for yourself to factor out the repeated code if you find yourself doing a lot of this; see [this](#)

`post` and `pd`. But beware: inheritance is not always the solution to everything. Often you can get away with something simple, like a type field and a `switch` statement, as shown:

```
1 // Inheritance isn't always the answer
2 //
3 // Instead of subclassing Honda, Mercedes, etc., we can often get away with
4 // switch statements or the like. rather than going with a class as an
5 // extremely heavyweight if/then statement.
6 //
7 // NOTE: in this particular example, we can actually replace the switch
8 // statement with an object lookup, but in general one might have more
9 // sophisticated logic in each switch case.
10
11 function Car(make, model) {
12     this.make = make;
13     this.model = model;
14 }
15 Car.prototype.toString = function () {
16     return "Make: " + this.make + " | Model: " + this.model;
17 };
18 Car.prototype.serviceIntervals = function() {
19     switch (this.make) {
20         case 'Honda':
21             out = [10000, 20000, 30000];
22             break;
23         case 'Mercedes':
24             out = [20000, 40000, 60000];
25             break;
26         default:
27             out = [5000, 10000, 20000];
28             break;
29     }
30     return out;
31 };
32
33 // Now let's instantiate some variables.
34
35 var accord = new Car('Honda', 'Accord');
36 var merc = new Car('Mercedes', 'S-Class');
37 var misc = new Car('Chrysler', 'Eminem Ride');
38
39 accord.toString();           // 'Make: Honda | Model: Accord'
40 merc.toString();            // 'Make: Mercedes | Model: S-Class'
41 misc.toString();            // 'Make: Chrysler | Model: Eminem Ride'
42 accord.serviceIntervals(); // [ 10000, 20000, 30000 ]
43 merc.serviceIntervals();   // [ 20000, 40000, 60000 ]
```

```
44 misc.serviceIntervals(); // [ 5000, 10000, 20000 ]
```

The lesson then is not to use inheritance when a switch statement, or an if/then will do instead. Here is another example where we use composition (`has_a`) rather than inheritance (`is_a`) to represent relationships between objects.

```
1 // Inheritance isn't always the answer, pt. 2
2 //
3 // Let's do another example where we show two different kinds of objects
4 // relate via composition.
5 function Wheel(isfront, isright) {
6     this.isfront = isfront;
7     this.isright = isright;
8 }
9 Wheel.prototype.toString = function () {
10    var pos1 = this.isfront ? 'f' : 'b'; // front/back
11    var pos2 = this.isright ? 'r' : 'l'; // right/left
12    return pos1 + pos2;
13 };
14 var wheel1 = new Wheel(true, true);
15 var wheel2 = new Wheel(true, false);
16 var wheel3 = new Wheel(false, true);
17 var wheel4 = new Wheel(false, false);
18
19 wheel1.toString(); // 'fr'
20 wheel2.toString(); // 'fl'
21 wheel3.toString(); // 'br'
22 wheel4.toString(); // 'bl'
23
24 // We use the invoke method in underscore.
25 // See underscorejs.org/#invoke
26 var uu = require('underscore');
27 function Car(make, model, wheels) {
28     this.make = make;
29     this.model = model;
30     this.wheels = wheels;
31 }
32 Car.prototype.toString = function () {
33     var jsdata = { 'make': this.make,
34                   'model': this.model,
35                   'wheels': uu.invoke(this.wheels, 'toString')};
36     var spacing = 2;
37     return JSON.stringify(jsdata, null, spacing);
38 };
39
40 var civic = new Car('Honda', 'Civic', [wheel1, wheel2, wheel3, wheel4]);
```

```

42  /*
43   { make: 'Honda',
44   model: 'Civic',
45   wheels:
46     [ { isfront: true, isright: true },
47       { isfront: true, isright: false },
48       { isfront: false, isright: true },
49       { isfront: false, isright: false } ] }
50 */
51
52 console.log(civic.toString());
53 /*
54 {
55   "make": "Honda",
56   "model": "Civic",
57   "wheels": [
58     "fr",
59     "fl",
60     "br",
61     "bl"
62   ]
63 }
64 */

```

Note that while you should know how to work with arrays and dictionaries by heart, it's ok to look up the syntax for defining objects as you won't be writing new classes extremely frequently.

Heuristics for OOP in JS

Here are a few tips for working with OOP in JS.

- *Consider using switches and types rather than inheritance:* You usually don't want to use inheritance if you can get away with a simple type field and switch statement in a method. For example, you probably don't want to make Honda and Mercedes subclasses of `Car`. Instead you'd do something like `car.model = 'Mercedes'`; combined with a method that uses the `switch` statement on the `car.model` field.
- *Consider using composition rather than inheritance:* You don't want to make a `Wheel` a subclass of `Car`. Conceptually, a `Car` instance would have a list of four `Wheel` instances as a field, e.g. `car.wheels = [wheel1, wheel2, wheel3, wheel4]`. A relationship does exist, but it's a composition relationship (`Car has_a Wheel`) rather than an inheritance relationship (`Wheel is_a Car`).
- *Use shallow hierarchies if you use inheritance:* In the event that inheritance really is the right solution, where both functions and behavior change too much to make type fields/switches reasonable, then go ahead but use a shallow inheritance hierarchy. A very deep inheritance hierarchy is often a symptom of using a class definition like an `if/else` statement.

- *Be careful about mutual exclusivity with inheritance.* A seemingly reasonable example would be `User` as a superclass and `MerchantUser` and `CustomerUser` as subclasses in an online marketplace app. Both of these `User` subclasses would have email addresses, encrypted passwords, and the like. But they would also have enough different functionality (e.g. different login pages, dashboards, verification flows, etc) that you'd actually be benefiting from inheritance. However, an issue arises if your Merchant users want to also serve as Customers, or vice versa. For example, Airbnb hosts can book rooms, and people who book rooms can serve as hosts. Don't box yourself into a corner with inheritance; try to look for truly mutually exclusive subclasses.
- *Think about classes and instances in terms of tables and rows, respectively.* In a production webapp most of your classes/instances will be respectively tied to tables/rows in the database via the ORM, and inheritance hierarchies with ORMs tend to be fairly shallow. This is the so-called [Active Record](#) paradigm.
- *Make your objects talk to themselves:* If you find yourself writing a function that is picking apart the guts of an object and calculating things, you should consider moving that function into an object method. Note how our `dbobj` object had a `toString` method that used the `this` keyword to introspect and print itself. Grouping code with the data it operates on in this way helps to manage complexity.
- *Verbally articulate your flow.* *Nouns are classes, verbs are functions or methods.* A good way to define the classes and functions for your new webapp is to write down a few paragraphs on how the app works. Any recurrent nouns are good candidates for classes, and any recurring verbs are good candidates for functions or methods. For example, in a Bitcoin app you might talk about users with different Bitcoin addresses sending and receiving payments. In this case, `User` and `Address` are good classes, and `send` and `receive` are good functions.

To summarize, then, you should now have a handle on a common suite of JS concepts that apply to both frontend JS (in the browser) and server-side JS (in node). These concepts include the basic global objects in JS interpreters and how to use them, techniques for functional programming (FP) in JS, and techniques for working with prototypes and objects to do something very similar to object-oriented programming (OOP) in JS.

Regulation, Disruption, and the Technologies of 2013

Overview

Perhaps the most common critique of the technology industry today is that too much money, ability, and energy is focused on social games, photosharing, advertising, todo lists, and the like. Some critics harken to a past where the Valley invested in tangible breakthroughs in PCs, semiconductors, and networking, others can't find much positive to say about technology in general, and generally many people feel that the Valley is now suffering from a failure of imagination (1, 2, 3, 4, 5, 6). A related and overlapping critique is that an innovation slowdown has occurred. This thesis has been most prominently advanced by Tyler Cowen in *The Great Stagnation* and by Founders Fund. While Cowen points to the macroscopic plateau in Total Factor Productivity, FF makes the empirical case for technological slowdown via three concrete examples in *What Happened to the Future*:

1. The travel speed from NY to London has actually decreased, with the retiring of the Concorde.
2. The cost of approving a drug has soared precipitously to \$3-4 billion.
3. The cost of electricity in cents-per-kilowatt-hour has actually increased.

Let's pull these observations together and see if we can advance a causal hypothesis. Is there some kind of force that is discouraging future entrepreneurs from working on substantive problems, especially in the physical world, despite their large market size? Or is it simply a question of a lack of ideas or bravery on the part of Silicon Valley?

Gaining Context

Before we answer in the abstract, let's begin with an assemblage of news clips to set the stage, a bit like Nicholson Baker's *Human Smoke*. There are quite a few of these, but by the end you will have some context and perhaps some hypotheses as to what's going on.

Transportation and Lodging

A four year NHTSA ban on self-driving cars ([CS Monitor](#)):

The last few years have seen self-driving cars go from the stuff of science fiction to the scientific method. Real cars are really driving themselves on some roads, and many current or near-future cars offer some form of assisted driving already. But the National Highway Traffic Safety Administration (NHTSA) wants to ban them.

Or, NHTSA wants each state to ban them, at least. In an announcement today outlining its policy on automated vehicles, NHTSA called for the ban of public use of self-driving cars. . . .

It's not yet clear how long NHTSA wants the ban to be in place; according to today's announcement, it could be a long, long ban:

"While the technology remains in early stages, NHTSA is conducting research on self-driving vehicles so that the agency has the tools to establish standards for these vehicles, should the vehicles become commercially available. The first phase of this research is expected to be completed within the next four years," the agency said in the statement.

James Fallows, Atlantic Editor, on Uber vs. Washington DC ([The Atlantic](#)):

As a longtime resident of DC, I am accustomed to misadventures in governance in our "taxation without representation" existence here. But a fight over a new competitor to the District's (often horrible) taxi service offers something I haven't seen in a while. Not routine retail-level corruption, nor skillful top-level favor trading, but instead what appears to be a blatant attempt to legislate favors for one set of interests by hamstringing another. I know, I know, this happens all the time – but the seeming crudity of this one gets my attention. . .

But it appears that the DC Council will vote today on a proposal to cripple Uber by ensuring that its minimum fare is five times higher than that for metered taxis, which also rules out a lower-cost hybrid option Uber has just introduced. C'mon!

Uber, Lyft, Sidecar and the CPUC ([LA Times](#)):

California Public Utilities Commission Chairman Michael R. Peevey sent a not-so-subtle message to the Los Angeles City Council on Tuesday: Hands off Uber, Lyft and Sidecar.

The commission had previously asserted jurisdiction over the companies and their ilk, which enable customers to use a smartphone app to summon rides from limousines (Uber) or private cars (Lyft, Sidecar, InstantCab). Peevey's proposed rule, which the full commission could consider in September, declares that such "transportation network companies" are "charter party passenger carriers," which are subject to the PUC's jurisdiction, and not taxis, which would be subject to local oversight.

Some members of the Los Angeles City Council have taken the opposite view, proposing an ordinance to regulate Web-enabled transportation companies as taxis.

Elon Musk of Tesla and Car Dealers ([Jalopnik](#)):

Another week, another victory for Tesla in their ongoing war with car dealerships. This time it's in North Carolina, whose legislature has backed off on a bill that would have blocked Tesla sales in that state.

That bill, backed by the N.C. Automobile Dealers Association, would have made the direct-to-customer sales model used by Tesla illegal in North Carolina. Though

it didn't mention Tesla by name, the bill was intended as a shot at the company, whose sales model threatens the traditional dealer-as-middleman concept that dominates how cars are sold.

The North Carolina bill made it through their senate, but the Raleigh News & Observer reports that House members weren't too keen on it - in part because of Tesla's strong quarterly profits, and because they liked the Model S after taking it for a test drive along with Gov. Pat McCrory.

John Carmack, co-founder of iD Software, lead programmer on Quake and Doom, and founder of Armadillo Aerospace ([link](#))

A couple things slowly brought me around to paying more attention. A computer game company doesn't need to have much to do with the government, but a company that flies rocket ships is a different matter. Due to Armadillo Aerospace, in the last decade I have observed and interacted with a lot of different agencies, civil servants, and congressmen, and I have collected enough data points to form some opinions.

My core thesis is that the federal government delivers very poor value for the resources it consumes, and that society as a whole would be better off with a government that was less ambitious. This is not to say that it doesn't provide many valuable and even critical services, but that the cost of having the government provide them is much higher than you would tolerate from a company or individual you chose to do business with. For almost every task, it is a poor tool.

So much of the government just grinds up money, like shoveling cash into a wood chipper. It is ghastly to watch. Billions and billions of dollars. Imagine every stupid dot-com company that you ever heard of that suckered in millions of dollars of investor money before leaving a smoking crater in the ground with nothing to show for it. Add up all that waste, all that stupidity. All together, it is a rounding error versus the analogous program results in the government. Private enterprises can't go on squandering resources like that for long, but it is standard operating procedure for the government.

Even if you could snap your fingers and get it, do you really want a razor sharp federal apparatus ready to efficiently carry out the mandates of whoever is the supreme central planner at the moment? The US government was explicitly designed to make that difficult, and I think that was wise.

So, the federal government is essentially doomed to inefficiency, no matter who is in charge or what policies they want it to implement. I probably haven't lost too many people at this point - almost nobody thinks that the federal government is a paragon of efficiency, and it doesn't take too much of an open mind to entertain the possibility that it might be much worse than you thought (it is).

Peers.org and Collaborative Consumption ([Politico](#)):

Founding partners include short-term apartment rental company Airbnb, car-sharing companies Getaround and Lyft, solar projects company Solar Mosaic and others. . . . PEERS "is a grass-roots organization that supports the sharing economy movement. We believe that by sharing what we already have - like cars,

homes, skills and time - everyone benefits in the process.” The group will be led by former Obama campaign and Democratic National Committee Digital Director Natalie Foster. Milicent Johnson, who founded a similar sharing advocacy group in the Bay area, will serve as director of partnerships and community building. Obama campaign veteran Alex Lofton will serve as managing director, while James Slezak will be director of strategy.

Drone Startups and the FAA ([Wired](#)):

Domestic-Drone Industry Prepares for Big Battle With Regulators . . .

“I didn’t change any guidelines,” Williams interrupted. “I didn’t say that any guidelines changed. I said that if a farmer as an individual wants to operate an unmanned aircraft according to the modeling rules, they can do that. The FAA rules are very clear about for-compensation and hire. If you’re going to operate an aircraft for compensation or hire, there’s a different set of rules that apply. So, you know, I’m not gonna split hairs over whether the farmer is making a profit or not, nor are we going to go look for him, but the bottom line is the rules are the rules and we have to enforce them until they’re changed.”

“So unmanned aircraft companies can operate R&D as long as they’re within the modeling guidelines?” Novara continued. Laughter and applause broke out among the hundreds of drone enthusiasts inside the Tyson’s Corner Ritz-Carlton.

“That’s why we have experimental certificates, to allow manufacturers—”

“The farmer doesn’t need an experimental certificate,” Novara pressed, “and everyone knows the experimental certificate process is available but not actually functional.”

Williams conceded that the current FAA rules “need to change,” since they were written for manned aircraft, “and that’s why we’re working hard to get the small unmanned aircraft rule out that will help resolve these issues. Until such time, we have to enforce the rules that are in place.”

“Is everyone else clear on this?” Novara asked, to bales of laughter. Some in the crowd shouted “No!” . . .

“If we were all smart guys, we’d be in consumer products, right?” Novara tells Danger Room. “It’s what I like doing. There’s just no money in it.”

AirBnB and New York State ([Guardian](#)):

What about the legal issues, in cities and countries where people aren’t allowed to rent out their rooms or properties without being licensed to do so? New York, Amsterdam and Quebec are just three high-profile examples of places where Airbnb has faced regulatory scrutiny.

Gebbia was bullish in his response. “When the car was introduced in 1908, people could experience a brand new way to travel that was more efficient than a horse and buggy,” he said.

“Can you believe that cities tried to outlaw cars in the United States? Can you imagine driving a car for a year then having to go back to a horse and buggy? The policy-makers adjusted to meet the demands of the people. We believe it’s

time for our invention, and it appears the world agrees, given we're in 40k cities in 192 countries."

Payments and Finance

Bart Chilton of CPUMC on Bitcoin ([Reuters](#)):

The top U.S. derivatives regulator is considering whether the Bitcoin virtual currency should be subject to its rules, a top official at the agency said.

Bart Chilton, one of five commissioners at the Commodity Futures Trading Commission, said he had asked staff to explore whether consumers needed more protection from any mishaps with Bitcoin, whose value collapsed last month.

"Here's what I know for sure: we could regulate it if we wanted. That is very clear," Chilton told Reuters in an interview on Monday. The Financial Times was first to report on Chilton's plans.

Bitcoin Foundation Response to Cease and Desist Letter ([Ars Technica](#), [Coindesk](#)):

According to the DFI's letter, which was sent on May 30, the Bitcoin Foundation requires licensure as a money transmitter under California law.

"The California state DFI said this was an invitation to talk. I've received nicer invitations, but we took it then as an opportunity to engage in a discussion about what we think the issues are and how we think the law agrees," Patrick Murck of the Bitcoin Foundation told CoinDesk.

The primary points raised in the response letter are that the foundation doesn't sell bitcoins nor operates in California, so it is not under the jurisdiction of the DFI.

"Even if we did operate an exchange or sell bitcoins, we have never done it with anyone in California, so they have no jurisdictional basis for coming and looking at us in the first place," said Murck. . .

Murck went on to say he thinks the response letter, which was drafted by legal firm Perkins Coie, will lead to more discussions with the DFI over the next few weeks, and will hopefully result in a letter of opinion.

Murck said attorneys are starting to become very popular among those in bitcoin and advised people with any concerns to seek legal advice. "It was great to walk into the Bitcoin London conference yesterday and have a legal panel that people love and pay so much attention to, but it's also a shame. "The people who should be front and centre on the stage are the entrepreneurs building companies. Hopefully we can get back to that place," he concluded.

Paypal and the Patriot Act ([CNN](#)):

EBay's PayPal accused of violating Patriot Act

PALO ALTO, California (Reuters) – A federal prosecutor has alleged eBay Inc. unit PayPal violated a 2001 anti-terror law aimed at fighting money laundering when it provided payment services to online gambling companies, the Web auctioneer said in its annual report filed Monday.

Eliot Spitzer and Paypal ([Who Killed Paypal](#)):

But PayPal's regulatory troubles persisted. The banking industry had tried and failed several times to set up competitors to PayPal and Billpoint. As entrenched industries often do, it turned to government when its efforts in the marketplace failed. Oregon, California, Illinois, and Louisiana subsequently sent Billpoint notices that it had failed to get a money transfer license. A director from the American Banking Association told CNET that online payment services should be classified and regulated as commercial banks - a move that likely would have killed off all online payment services except those run by existing banks.

More class actions followed. New York Attorney General Eliot Spitzer cited PayPal for posting a user agreement that "wasn't clear enough." He also subpoenaed all documents pertaining to PayPal's use in online gaming sites, suggesting the company was in violation of New York gambling laws. Spitzer's investigation was followed by a U.S. Justice Department determination that PayPal's use by gaming sites was a violation of the USA PATRIOT Act.

The financial pressures of battling aggressive government officials and opportunistic class action lawyers, all while trying to stave off a better-funded competitor, soon became too much for the still-young company to bear. "It was clear," Jackson writes, "that PayPal now faced many challenges outside the marketplace. Entrepreneurial nimbleness may have helped us survive the company's post-merger internal turmoil and Billpoint's fierce competitive charge, but these new threats would require a different approach."

In July 2002 PayPal executives sold the start-up firm to their longtime nemesis, eBay. Jackson notes that the sale had some obvious benefits. The company's new parent already had a formidable, well-funded legal team in place to deal with PayPal's litigation and regulation troubles. Also, eBay promised to do away with Billpoint, essentially securing PayPal's position as the premier online payment provider.

Online Poker and UIGEA ([PDF](#)):

The business of online gambling spans the globe and touches every corner of the United States. Worldwide, online gambling is increasingly a legal and regulated activity that generates almost \$30 billion of revenue a year. In the United States, public policy on the subject has been schizophrenic. Online gambling is presently being conducted domestically for pari-mutuel betting on horse races and for state lotteries, yet government policy has been hostile to otherforms of online gambling, and has included criminal prosecutions of online gambling operators and their payment processing partners. Despite this government opposition, millions of Americans spend \$4 billion every year to gamble online. Prosecutions against online gambling operators have driven the more responsible offshore operators out of the U.S. market, leaving Americans to conduct their online gambling through largely unregulated websites.

In contrast, about 85 nations have chosen to legalize and regulate online gambling. Numerous Western nations - including the United Kingdom, France, Italy, and some provinces in Canada - have created structuresfor tight regulation of the online

gambling industry. This course provides consumer protections for individuals while also generating jobs, economic opportunity and government revenue.

SEC vs. Crowdfunding ([Morrill, Crowdvalley](#)):

On March 26th The Funders Club received a no-action letter from the Securities and Exchange Commission stating that it will not recommend enforcement action against the year-old private equity investment platform, making it the first government sanctioned online VC.

... AngelList, a competing service used by early stage companies to receive introductions to investors and take some online investment, received its own no-action letter just two days later on March 28th.

Goldman, Facebook, and the SEC ([Wired](#)):

Goldman-Facebook Deal Draws SEC Scrutiny of Startup Investing (Updated)

Goldman Sachs' \$450 million investment in social networking juggernaut Facebook has prompted a federal inquiry into whether the deal is designed to avoid rules aimed at protecting investors, The Wall Street Journal reported, citing "people familiar with the situation."

Biotech

Case 5:10-cv-04018-MWB Document 11-1 Filed 04/26/10 Page 2 of 30

a. There is No Right to Consume or Feed Children Any Particular Food.	25
b. There is No Generalized Right to Bodily and Physical Health.	26
c. There is No Fundamental Right to Freedom of Contract....	27
5. FDA's Regulations Rationally Advance The Agency's Public Health Mission.....	27

Figure 1: The FDA has officially taken the position that "There is No Right to Consume or Feed Children Any Particular Food", "There is No Generalized Right to Bodily and Physical Health", and "There is No Fundamental Right to Freedom of Contract" and that its regulations "Rationally Advance the Agency's Public Health Mission". (Source: [FDA US District Court Brief, pages 25-27](#)).

FDA: “There is No Generalized Right to Bodily or Physical Health” ([FDA Legal Filing](#)):

FDA: There is No Generalized Right to Bodily and Physical Health.

Plaintiffs’ assertion of a “fundamental right to their own bodily and physical health, which includes what foods they do and do not choose to consume for themselves and their families” is similarly unavailing because plaintiffs do not have a fundamental right to obtain any food they wish. In addition, courts have consistently refused to extrapolate a generalized right to “bodily and physical health” from the Supreme Court’s narrow substantive due process precedents regarding abortion, intimate relations, and the refusal of lifesaving medical treatment.

See Glucksberg, 521 U.S. at 721 (warning that the fact “[t]hat many of the rights and liberties protected by the Due Process Clause sound in personal autonomy does not warrant the sweeping conclusion that any and all important, intimate, and personal decisions are so protected”); see also Cowan v. United States, 5 F. Supp. 2d 1235, 1242 (N.D. Okla. 1998) (rejecting a claim that the plaintiff had the fundamental “right to take whatever treatment he wishes due to his terminal condition regardless of whether the FDA approves the treatment”).

Reputation and Power, Chapter 1 ([PDF](#))

None of this import was lost on Genentech’s executive at the time, G. Kirk Raab. Raab was hired specifically to smooth the company’s journey through the regulatory process. Years later, Raab would describe regulatory approval for his products as the fundamental challenge facing his company. And he would depict the Administration in a particularly vivid metaphor.

I've told a story hundreds of times to help people understand the FDA. When I was in Brazil I worked on the Amazon River for many months selling Terramycin for Pfizer. I hadn't seen my family for eight or nine months. They were flying in to Sao Paulo, and I was flying down from some little village on the Amazon to Manous and then to Sao Paulo. I was a young guy in his twenties. I couldn't wait to see the kids. One of them was a year-old baby, the other was three. I missed my wife.

There was a quonset hut in front of just a little dirt strip with a single engine plane to fly me to Manous. I roll up and there is a Brazilian soldier standing there. The military revolution had happened literally the week before. So this soldier is standing there with this machine gun and he said to me: "You can't come in." I was speaking pretty good Portuguese by that time. I said: "My god, my plane, my family, I gotta come in!" He said again: "You can't come in." I said: "I gotta come in!" And he took his machine gun, took the safety off, and pointed it at me, and said: "You can't come in." And I said: "Oh, now I got it. I can't go in there."

And that's the way I always describe the FDA. The FDA is standing there with a machine gun against the pharmaceutical industry, so you better be their friend rather than their enemy. They are the boss. If you're a pharmaceutical firm, they own you body and soul.

... In practice, dealing with the fact of FDA power meant a fundamental change in corporate structure and culture. At Abbott and at Genentech, Raab's most central transformation was in creating a culture of acquiescence toward a government agency. As was done at other drug companies in the late twentieth century, Raab essentially fired officials at Abbott who were insufficiently compliant with the FDA.

Harvard's Daniel Carpenter ([Princeton Press](#)):

The U.S. Food and Drug Administration is the most powerful regulatory agency in the world. How did the FDA become so influential? And how exactly does it wield its extraordinary power? Reputation and Power traces the history of FDA regulation of pharmaceuticals, revealing how the agency's organizational reputation has been the primary source of its power, yet also one of its ultimate constraints.

Daniel Carpenter describes how the FDA cultivated a reputation for competence and vigilance throughout the last century, and how this organizational image has enabled the agency to regulate an industry as powerful as American pharmaceuticals while resisting efforts to curb its own authority.

Google fined \$500 million by the FDA ([NYT](#)):

The Justice Department's settlement of a criminal investigation of Google for allowing Canadian pharmacies to advertise drugs for distribution in the United States reflected an effort by prosecutors to extend the reach of federal drug laws. This may present future challenges to Internet search companies over their advertisements.

ONC and FDA clash over EHR Regulation ([iHealthBeat](#)):

Under the direction of National Coordinator for Health IT David Blumenthal, ONC has pushed for greater adoption of EHRs by health care providers.

FDA, charged with ensuring the safety and effectiveness of medical devices, has called for greater regulation of EHR systems. In February, Jeffrey Shuren – head of FDA's medical devices division – linked six deaths and more than 200 injuries to health IT problems, based primarily on voluntary reports to FDA.

ONC has dismissed Shuren's findings as "anecdotal and fragmentary." EHR proponents contend that excessive regulation could stifle EHR innovation and inhibit EHR adoption. Proponents also state that, despite possible errors, EHRs are generally safer than paper records.

Drug Shortages and Regulation ([Health Affairs](#)):

Output Controls. The Federal Food and Drug Administration (FDA) has been stepping up its quality enforcement efforts - levying fines and forcing manufacturers to retool their facilities both here and abroad. Not only has this more rigorous regulatory oversight slowed down production, the FDA's "zero tolerance" regime is forcing manufacturers to abide by rules that are rigid, inflexible and unforgiving.

For example, a drug manufacturer must get approval for how much of a drug it plans to produce, as well as the timeframe. If a shortage develops (because, say, the FDA shuts down a competitor's plant), a drug manufacturer cannot increase its output of that drug without another round of approvals. Nor can it alter its timetable production (producing a shortage drug earlier than planned) without FDA approval.

Even the Drug Enforcement Agency (DEA) has a role - because minute quantities of controlled substances are often used to make other drugs. This is the apparent reason for a nationwide shortage of ADHD drugs, for example, including the generic version of Ritalin. And like the FDA, DEA regulations are rigid and inflexible. For example, if a shortage develops and the manufacturers have reached their preauthorized production cap, a manufacturer cannot respond by increasing output without going back to the DEA for approval.

The Centers for Medicare and Medicaid Services (CMS) also has a role levying large fines for "overcharging," forcing some companies to leave the generic market altogether.

Roche statement on LDTs, including "Specific Suggested Changes to the Draft Guidance by Line Number" ([regulations.gov](#), [pdf](#))

Roche is becoming increasingly concerned about FDA's continued lack of oversight of LDTs. This lack of oversight, particularly when a cleared or approved IVD is available, has created a dichotomy in which manufacturers of diagnostic tests invest vast resources to research, develop, verify and validate innovative new tests that are subject to intense FDA scrutiny, while clinical laboratories develop and offer LDTs that are neither supported by clinical data nor subject to FDA's rigorous review . . . These differential standards, especially in companion diagnostics, generate unnecessary and unacceptable risks for physicians and patients who count on companion diagnostics to direct therapy, and create an uneven playing field for manufacturers of cleared or approved IVD companion diagnostics.

Appendix: Specific Suggested Changes to the Draft Guidance by Line Number

Appendix

Specific Suggested Changes to the Draft Guidance by Line Number

Comment Number	Section	Line No	Proposed Change	Comment/Rationale
1	II	270-271	FDA should clarify requirements for IVD companion diagnostics for generics, specifically, that no separate IVD companion diagnostic submission would be needed for a generic form of the companion therapeutic.	The Draft Guidance document says that instructions for the use of an IVD companion diagnostic should be listed in the labeling of the therapeutics <i>and any generic equivalent</i> but no specifics are given. No separate submission should be required for generic equivalents.
2	II	280-286	FDA should clarify Footnote 4 to make clear when a companion diagnostic and its therapeutic constitute a Combination Product. The role of the Office of Combinations Products should be clarified.	We believe designation of companion diagnostics as combination products is generally inappropriate and unnecessary - in part because the Primary Mode of Action is already known; and thus there is no need for the Office of Combination Products to make that determination. An exception might be with any companion diagnostic that would be linked directly to a therapeutic intervention without physician intervention / interpretation prior to administration of the therapy. These should be considered combination products. An example would be a blood glucose monitoring insulin pump.

Figure 2: Roche's specific line-item edits to proposed FDA regulation, by line number (*Source: Roche regulations.gov filing*)

FDA and DNA Dilemma Interview ([Newsweek](#)):

Newsweek: What exactly would constitute a “medical claim?” Would pointing people to medical research papers [qualify]?

FDA: It depends. There are rules as to how one can do that - Those rules are actually worked out pretty well, and they just would need to make sure they’re staying within the rules.

Newsweek: Are those rules on the Web?

FDA: I don’t know where the policy is. I would have to get it for you. It’s an agencywide policy. I would have to find it for you. And it won’t be that easy for people to follow it...

FDA on information security ([Washington Post](#)):

The agency has urged hospitals to allow vendors to guide them on security of sophisticated devices. But the vendors sometimes tell hospitals that they cannot update FDA- approved systems, leaving those systems open to potential attacks. In fact, the agency encourages such updates.

“A lot of people are very confused about FDA’s position on this,” said John Murray Jr., a software compliance expert at the agency.

The Park Doctrine ([Merola](#)):

In *United States v. Park*, the Supreme Court held that a responsible corporate official can be convicted of a misdemeanor based on his or her position of responsibility and authority to prevent and correct violations of the Food Drug and Cosmetic Act (FDCA). Thus, evidence that an individual participated in the alleged violations or even had knowledge of them is not necessary.

FDA issued one [Form 483](#) every 50 minutes in 2011 ([FDAZilla](#)):

Based on new data obtained by FDAZilla, the FDA should surpass 10,000 483s in 2011 for the second time in its history, breaking its all-time record for the third consecutive year. In 2010, the FDA issued 10,437 483s, breaking the previous high-water mark of 9,666 in 2009. Through November 2, 2011, the FDA has issued 9,052 483s, approximately 215 more 483s than the same time period in 2010, roughly a 2.4% increase. Extrapolating for the rest of the year, we expect the FDA to issue roughly 10,690 483s for 2011. That’s one 483 every 50 minutes.

UltraRad’s UltraPACS device labeled “adulterated”, in part for insufficient revision history of Microsoft Sharepoint ([fda.gov](#)):

This inspection revealed that this device is adulterated within the meaning of section 501(h) of the Act...

Off-the-shelf software (Microsoft SharePoint) is being used by your firm to manage your quality system documents for document control and approval. However, your firm has failed to adequately validate this software to ensure that it meets your

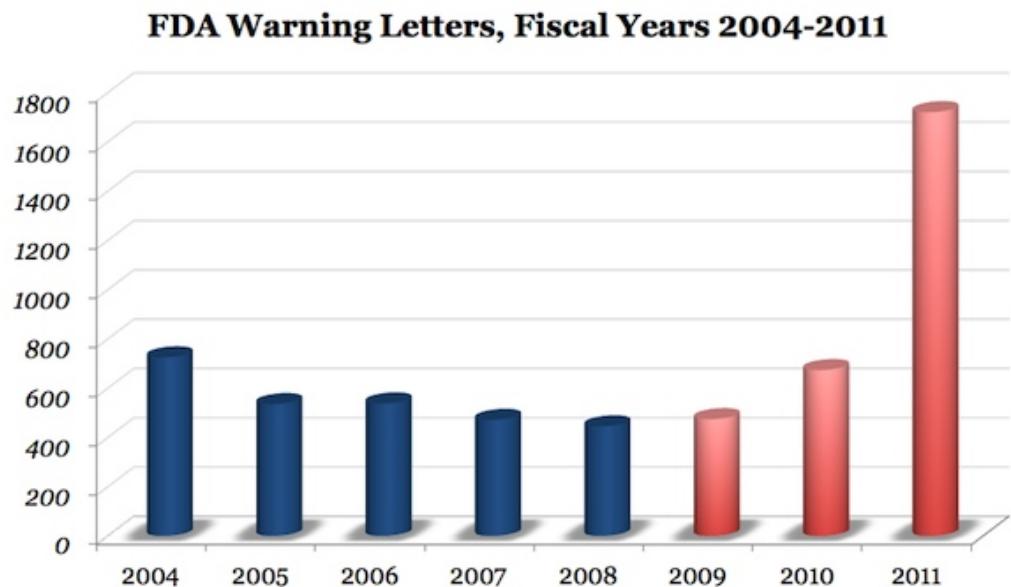


Figure 3: Historical trend in FDA Warning Letters.

needs and intended uses. Specifically, at the time of this inspection there were two different versions of your CAPA & Customer Complaint procedure, SOP-200-104; however, no revision history was provided on the SharePoint document history. Your firm has failed to validate the SharePoint software to meet your needs for maintaining document control and versioning. . . .

You should take prompt action to correct the violations addressed in this letter. Failure to promptly correct these violations may result in regulatory action being initiated by FDA without further notice. These actions include, but are not limited to, seizure, injunction, and/or civil money penalties.

Drug Approvals Cost \$4B on Average ([Matthew Herper](#)):

The Truly Staggering Cost Of Inventing New Drugs

During the Super Bowl, a representative of the pharmaceutical company Eli Lilly posted the on the company's corporate blog that the average cost of bringing a new drug to market is \$1.3 billion, a price that would buy 371 Super Bowl ads, 16 million official NFL footballs, two pro football stadiums, pay of almost all NFL football players, and every seat in every NFL stadium for six weeks in a row. This is, of course, ludicrous.

The average drug developed by a major pharmaceutical company costs at least \$4 billion, and it can be as much as \$11 billion.

Mobile Health and the FDA ([Entrepreneur](#)):

When the U.S. Food & Drug Administration sent a warning letter to an Indian app developer in late May, tech entrepreneurs in this country took notice. The FDA warned Biosense Technologies Private Ltd. that its app - which is designed

to work with a urine-testing kit - is actually a medical device, and therefore it must be cleared by the agency.

The large and growing community of app developers doesn't expect this to be the last time the FDA weighs in on mobile apps marketed for health-related uses. "There are millions of medical apps out there. The industry is concerned," says Gabriel Vorobiof, a Los Angeles cardiologist and co-founder of PadInMotion, a New York company developing mobile tools for hospital use. "It's just not clear how far [the FDA] will go."

The Lauren Stevens Case ([MJ](#), [IC](#), [pdf](#)):

MJ: No matter what they say, or don't say, there must be people in the Justice Department who wish they had never heard of Lauren Stevens, the former GlaxoSmithKline vice president and counsel who was acquitted on Tuesday of lying to the federal government about GSK's marketing of a drug.

Prosecutors made tactical mistakes, to be sure. But far worse than that, they aggressively pushed a case that never should have been brought in the first place, a federal judge concluded as he took the unusual step of acquitting Stevens before the jurors could even start deliberating.

It would be a miscarriage of justice to permit this case to go to the jury; U.S. Judge Roger Titus of the District of Maryland ruled in tossing out the six charges: one count of obstructing an official proceeding, one of concealing and falsifying documents and four of lying to the Food and Drug Administration. "I conclude on the basis of the record before me that only with a jaundiced eye and with an inference of guilt that's inconsistent with the presumption of innocence could a reasonable jury ever convict this defendant" Titus declared.

IC: It's not often that we in-house lawyers get a scare like United States v. Lauren Stevens. Since the trial ended on May 10 when the judge granted a motion for acquittal, and the jury reportedly stood up and applauded, law firm advisories, blogs and seminars on how in-house lawyers can avoid going to jail for making statements to the government have been pouring forth.

Mobile MIM and the FDA ([link](#)):

Two of our lead engineers, Jeremy Brockway and Dave Watson, downloaded XCode, learned Objective C, and built a prototype, all on their off hours, and in only one week. It was remarkable. We saw the CT scan on the iPhone, scrolled through slices, and realized that everything had just changed. . . .

We were one of eleven developers that presented during the [Apple WWDC] keynote. That week we won an Apple Design Award for Best iPhone Healthcare & Fitness Application.

Within only a few weeks of submitting, we were contacted by the FDA and told that our app could not be on the app store (despite the fact that it was both free and labeled as "not intended for diagnostic use") because it served as marketing for a device that was not cleared for marketing. We promptly removed it. . . .

Then, over the next few months, we discovered that our proposed device raised more questions than we had anticipated. In order to make their determination, the FDA wanted more information than we had provided. The process stalled out as we reviewed what we would have to do next. This 510(k) was declared not substantially equivalent (NSE) because of insufficient data. . . .

To be honest, this dramatically new direction for our company, and the speed at which it occurred, left us ill-prepared for the scope of the regulatory process that would unfold.

The “It Has Come to our Attention” Letter ([Eye on FDA](#)):

There are Warning Letters, Untitled Letters, otherwise known as Notice of Violation Letters, and now there is the “It Has Come to Our Attention Letter”. I have to admit, I had not heard of this particular kind of letter before, but one was sent this month the maker of a medical app that performed the task of urine analysis. I did end up finding a few other examples, however, this was the only one I found that was actually had that as a title at the top - “It Has Come to Our Attention Letter”.

In any case, this “It Has Come to Our Attention Letter” - hereafter IHCOAL - raised the fact that FDA became aware of the existence of an app that served as a device allowing one to perform one’s own urine analysis with the assistance of a smart phone. . . . In that case, then FDA said there needs to be clearance for the whole new system of analysis - the app used in conjunction with the strips.

“Bold move to make themselves noticed. . . brought them to our attention” ([GenomeWeb](#)):

Lastly, ignore FDA’s sudden and questionable interest in a private company’s marketing budget. As OIVD Director Alberto Gutierrez described it this week to my colleague Turna Ray, “[t]he fact is that Pathway’s bold move to make themselves noticed achieved its end and brought them to our attention.”

Labcorp, FDA, and Ovasure ([MDT](#)):

In an Oct. 20 letter to FDA, LabCorp defends its decision to launch OvaSure out of its nationwide lab network in June without FDA clearance or approval, but also announced plans to discontinue the product as of Oct. 24 to maintain “positive and responsible relationships with regulatory agencies.” LabCorp is requesting a meeting with FDA’s Office of In Vitro Diagnostic Device Evaluation and Safety (OIVD). . . .

The enforcement action is unusual since FDA typically does not exert authority over “home brew” tests used as a lab service. Instead, CMS regulates lab processes and protocols under the Clinical Laboratory Improvement Amendments.

Antitrust and Acquisitions

Instagram Antitrust Scrutiny in the US and UK ([ZDNet](#)):

You'd think it would be simple: Facebook wants to buy Instagram, the parties agree on a price, Facebook writes a big fat check, Facebook owns Instagram, now you can follow Justin Bieber and apply special effects to photos of your cat, using both services, seamlessly, whether you're at your desk or on your phone.

Nothing is simple, however, when the antitrust regulators get involved.

Under the Hart-Scott-Rodino Antitrust Improvements Act of 1976 (HSR), any purchase over \$68.2 million requires a detailed filing with the Federal Trade Commission (FTC) and Department of Justice (DOJ). This number gets adjusted each year relative to GDP.

In the case of Facebook's acquisition of Instagram, a \$1 billion transaction, the FTC has now served Facebook with a "second request," in other words Facebook must now produce a mountain of documents that an army of government lawyers will review to make sure the purchase of Instagram didn't violate any antitrust laws.

Antitrust, Blockbuster, Hollywood Video, and Netflix ([Motley Fool](#)):

Little more than a year ago, a combination between the largest movie rental company, Blockbuster (NYSE: BBI), and the second-largest, Hollywood Entertainment (Nasdaq: HLYW), would have certainly raised antitrust questions. Blockbuster has more than 9,000 retail locations worldwide and would combine with Hollywood Video's nearly 2,000 locations to simply dominate the industry.

But things still change quickly in Internet time, which is why antitrust considerations on Blockbuster's \$700 million offer for Hollywood Entertainment, should they come up at all, would be seriously wrongheaded. The retail movie rental business has been thrown into turmoil by the emergence of Netflix (Nasdaq: NFLX), the online mail-order DVD rental company.

When I say "would have raised antitrust questions," I'm not guessing. More than five years ago, the Federal Trade Commission scotched a planned merger between the companies on the basis that it concentrated too much of the industry's volume in one company.

Complete Genomics and Antitrust ([NYT](#)):

The Chinese firm, BGI-Shenzhen, said in a statement this weekend that its acquisition of Complete Genomics, based in Mountain View, Calif., had been cleared by the federal Committee on Foreign Investment in the United States, which reviews the national security implications of foreign takeovers of American companies. The deal still requires antitrust clearance by the Federal Trade Commission. . . .

Much of the alarm about the deal has been raised by Illumina, a San Diego company that is the market leader in sequencing machines. It has potentially the most to lose from the deal because BGI might buy fewer Illumina products and even become a competitor. Weeks after the BGI deal was announced, Illumina made its own belated bid for Complete Genomics, offering 15 cents a share more than BGI's bid of \$3.15. But Complete Genomics rebuffed Illumina, saying such a merger would never clear antitrust review. . . .

BGI and Complete Genomics point out that Illumina has long sold its sequencing machines - including a record-setting order of 128 high-end machines - to BGI without raising any security concerns. Sequencing machines have not been subject to export controls like aerospace equipment, lasers, sensors and other gear that can have clear military uses.

Government

GovExec.com About page (govexec.com):

Government Executive's essential editorial mission is to cover the business of the federal government and its huge departments and agencies - dozens of which dwarf the largest institutions in the private sector. We aspire to serve the people who manage these huge agencies and programs in much the way that Fortune, Forbes, and Business Week serve private-sector managers.

China and the USG ([Washington Post](http://www.washingtonpost.com)):

The information compromised by such intrusions, security experts say, would be enough to map how power is exercised in Washington to a remarkably nuanced degree. The only question, they say, is whether the Chinese have the analytical resources to sort through the massive troves of data ...

"They're trying to make connections between prominent people who work at think tanks, prominent donors that they've heard of and how the government makes decisions," said Dan Blumenthal, director of Asian studies at the American Enterprise Institute, which also has been hacked. "It's a sophisticated intelligence-gathering effort at trying to make human-network linkages of people in power, whether they be in Congress or the executive branch."

Matthew Yglesias on Food Trucks ([Slate](http://slate.com)):

An existing provision of the San Francisco municipal code, for example, states that any business may comment on an application for a new vending license and directs the city to "consider" whether the proposed operation is located within 300 feet of a business that sells the same type of food or merchandise. It would be as if Slate were allowed to complain that it should be illegal to launch a new website to compete with our offerings, and that government should take our complaint seriously. . . .

It's difficult to know precisely where the line should be drawn. The food service industry is generally heavily regulated for safety purposes, and trucks should be no exception to that. And food sales are intimately related to parking, a fraught and much-regulated activity all its own. But a basic rule of thumb seems to suggest itself: The fact that business owners would prefer not to face competition is not a valid regulatory purpose. A food truck is a kitchen and a vehicle and should need to follow the rules that generally apply to both thing.

Most of all, the fact that an existing business owner objects to the practices of a new business is a terrible reason to block a truck from operating. Space is scarce and rents are high in the centers of major American cities. If new competition can bring prices down, we'll all be better off in the long run.

Matthew Yglesias on Small Business ([Slate](#)):

And thus I became a small-business man. Or, rather, I'm becoming one. Entrepreneurship - even on the smallest and most banal scale - turns out to be a time-consuming pain in the you-know-what. My personal inconveniences aren't a big deal, but in the aggregate, the difficulty of launching a business is a problem and it may be a more important one as time goes on.

In the District of Columbia, I need to get a simple Basic Business License to rent out a single dwelling. After puzzling over the Department of Consumer and Regulatory Affairs website for a bit, it became clear that step No. 1 was actually to file form FR-500 with the Office of Tax and Revenue, which you can do online. Then it was time to hustle down to the DCRA (which closes at 4:30 p.m.) to file the paperwork. Once there, I learned that filing the FR-500 online wasn't good enough - I needed a hard copy. Fortunately, the Office of Tax and Revenue was right across the street, so I went there and refiled. Then it was back to the DCRA to stand in line to get a number, wait for the number to be called, do some more paperwork, wait in another line for the cashier, fork over \$100 in fees, then get a slip from the cashier to finalize the paperwork.

But then it turned out I needed to go to a third office, the Rental Accommodations Division of the Department of Housing and Community Development. It closes at 3:30 in the afternoon and required a 15-minute walk through a sketchy neighborhood. So the next morning I went down to that Rental Accommodations office to file a paper claiming exemption from D.C.'s rent control law.

The striking thing about all this isn't so much that it was annoying - which it was - but that it had basically nothing to do with what the main purpose of landlord regulation should be - making sure I'm not luring tenants into some kind of unsafe situation.

Microsoft and K-Street ([Tim Carney](#)):

But it grated on Hatch and other senators that Gates didn't want to want to play the Washington game. Former Microsoft employee Michael Kinsley, a liberal, wrote of Gates: "He didn't want anything special from the government, except the freedom to build and sell software. If the government would leave him alone, he would leave the government alone."

This was a mistake. One lobbyist fumed about Gates to author Gary Rivlin: "You look at a guy like Gates, who's been arrogant and cheap and incredibly naive about politics. He genuinely believed that because he was creating jobs or whatever, that'd be enough."

Gates was "cheap" because Microsoft spent only \$2 million on lobbying in 1997, and its PAC contributed less than \$50,000 during the 1996 election cycle.

"You can't say, 'We're better than that,' " a Microsoft lobbyist told me on Friday. "At some point, you get too big, and you can't just ignore Washington."

"You can sit there and say, 'We despise Washington and we don't want to have anything to do with them,' " the lobbyist said. "But guess what? We're going to have hearings about the [stuff] you do." . . .

Microsoft now plays ball in Washington, and Orrin Hatch's public flogging of Gates was a major reason. "It's been a year since I was in D.C.," Gates wrote the night before his Hatch hearing. "I think I'm going to be making this trip a lot more frequently from now on."

Eric Schmidt, Google Chairman and former CEO ([Washington Post](#)):

And one of the consequences of regulation is regulation prohibits real innovation, because the regulation essentially defines a path to follow - which by definition has a bias to the current outcome, because it's a path for the current outcome. . . .

So what happened was that there was something called the Clipper chip, which was the attempt by the government to enforce encryption on a particular communications aspect. And this was 1994. And it was the first time I know of that the Valley organized around a stupid technological thing that was going to be forced on us. . . . All of us spent a lot of time and we eventually defeated it, but I think for many people that was sort of a wake-up call that the government could actually pass a law that was stupid, that would actually do something wrong and wouldn't work. . . .

And then we had the bubble, so all of a sudden the politicians showed up. We thought the politicians showed up because they loved us. It's fair to say they loved us for our money. And this was before caps were in place, so there's this huge fundraising cycle in the late 90s. Republican and Democrat by the way. Everyone fed at the trough of money.

But at the time, we took the position of 'hands off the Internet.' You know, leave us alone. And that's probably still the general view here. The government can make regulatory mistakes that can slow this whole thing down, and we see that and we worry about it.

Steve Jobs on US Factories ([NYT](#), [Politico](#)):

But as Steven P. Jobs of Apple spoke, President Obama interrupted with an inquiry of his own: what would it take to make iPhones in the United States?

Not long ago, Apple boasted that its products were made in America. Today, few are. Almost all of the 70 million iPhones, 30 million iPads and 59 million other products Apple sold last year were manufactured overseas.

Why can't that work come home? Mr. Obama asked. Mr. Jobs's reply was unambiguous. 'Those jobs aren't coming back,' he said, according to another dinner guest.

. . . According to Walter Isaacson's biography of Jobs, he expressed admiration for Chinese business practices and decried U.S. regulations and labor rules.

Peter Thiel and Eric Schmidt ([CNN](#), [MR](#)):

PETER THIEL: But, if we're living in an accelerating technological world, and you have zero percent interest rates in the background, you should be able to invest all of your money in things that will return it many times over, and the

fact that you're out of ideas, maybe it's a political problem, the government has outlawed things. But, it still is a problem. . . .

ERIC SCHMIDT: What you discover in running these companies is that there are limits that are not cash. There are limits of recruiting, limits of real estate, regulatory limits as Peter points out. There are many, many such limits. And anything that we can do to reduce those limits is a good idea.

Chris Dixon of A16Z ([Blog](#)):

A common way to think of business regulations is by analogy to sports: the rules are specified up front, and the players follow the rules. But real regulations don't work that way. Regulations follow business as much as business follows regulations.

Sometimes the businesses that change regulations are startups. Startups don't have the resources to change regulations through lobbying. Instead, they need to start with regulatory hacks: "back door" experiments that demonstrate the benefits of their ideas. With luck, regulators are forced to follow. . . .

Nextel was one of the all-time great regulatory hacks. In the late 80s and early 90s, the FCC's rules banned more than two cellular operators per city. As Nextel's cofounder said, "the FCC thought a wireless duopoly was the perfect market structure". Nextel (called Fleet Call at the time) circumvented these rules by acquiring local (e.g. taxi, pizza truck) dispatch radio companies, which they then connected to create a nationwide (non-dispatch) cell phone service.

Predictably, the cellular incumbents tried to regulate Nextel out of existence. . . . The incumbents argued that Nextel's service would interfere with public safety frequencies and therefore endanger the public. They also argued that Nextel's service would be too expensive . . . And their call quality would be inferior . . . The FCC eventually decided not to block Nextel. Nextel grew to become a top five US cellular operators before it was acquired by Sprint in 2004 for \$35B. Their service turned out to be cost-competitive, high quality, and safe. The only thing endangered were the incumbents' profits.

Of course regulations that truly protect the public interest are necessary. But many regulations are created by incumbents to protect their market position. To try new things, entrepreneurs need to find a back door. And when they succeed, it will all look obvious in retrospect. Today's regulatory hack is tomorrow's mainstream industry.

Alex MacCaw, Stripe/Twitter Engineer and O'Reilly Author ([Blog](#)):

Paradoxically, regulation can sometimes result in innovation. Within highly regulated industries, you often get entrenched businesses and monopolies ripe for disruption. While regulation and red tape may strangle fledgling startups, if they can struggle through and overcome it they have a distinct advantage - less competition.

What's more, regulation is often 'hackable'. Look hard, and you'll see lots of startups out there flirting with grey areas and testing the boundaries. Often it requires

a startup environment to think outside the box when it comes to interpreting and applying regulations. Most don't like to talk about it for obvious reasons.

The are three industries that are great examples of this phenomena at work: drones, bio-tech and payments. . .

A Summary

All right. That's a fair bit of material, but the reason we chose so many articles is to illustrate that the phenomenon we are about to discuss is:

- *Not limited to low-quality companies.* Note the seniority and technological accomplishments of many of the individuals involved: [Schmidt](#), [Thiel](#), [Jobs](#), [Carmack](#), [Musk](#), and so on.
- *Not limited to particular industries.* The articles cover [food](#), [planes](#), [cars](#), [drugs](#), [devices](#), [sequencing](#), [medical records](#), [hotels](#), [space travel](#), [payments](#), [investment](#), and even [online poker](#) and [photosharing](#).
- *Not limited to a particular political orientation.* [Natalie Foster](#) and [Matthew Yglesias](#) are Democrats; [Orrin Hatch](#) and the [North Carolina and Texas Senate](#) are Republican.

With those points made up front, let's see if we can summarize a few points for the prospective entrepreneur:

1. *Understand regulation.* Any business in the US that involves the physical world ([1](#), [2](#), [3](#)), and many businesses in the digital world ([1](#), [2](#), [3](#)), will mean concerning yourself with regulation at the local, state, and federal levels.
2. *Non-US businesses may have more latitude.* If you are outside the US, you may have much more latitude ([1](#), [2](#)). However, you may still be directly subject to US regulations if you serve US customers, or else indirectly subject if your local regulatory agency has “harmonized” its regulatory structure with its US counterpart.
3. *DC is not civics 101.* Washington does not necessarily work like [civics 101](#), where the only players are the executive, legislative, and judiciary. Academia, the press, nonprofits, and regulatory agencies all play crucial roles in “how the government makes decisions” ([1](#), [2](#)).
4. *Agencies are active, not passive.* Regulatory agencies are active entities and experience turf conflicts between themselves ([1](#), [2](#), [3](#), [4](#)).
5. *Agency interpretations may be found illegal.* Regulatory agencies have interpretations of the law and their authorities under the law that do not always hold up in court ([1](#), [2](#), [3](#)).
6. *News attracts regulation.* The attention of a regulator is often a function of a company’s prominence in the news media ([1](#), [2](#), [3](#))
7. *Legacy businesses often use regulations in their favor.* From restaurants and taxis to car dealers and MNC pharmas, legacy businesses use regulation to erect barriers to entry ([1](#), [2](#), [3](#), [4](#))

8. *Regulators are police agencies.* Regulatory agencies have the power to impose fines and mount raids (1, 2, 3), and have budgets that are comparable to huge Fortune 500 companies (1, 2)
9. *Travel or technology can change jurisdiction.* Overseas travel (1, 2) or modern technology can often obviate an existing regulation or render its applicability in doubt, pending a court case.
10. *Understand what regulation means for you.* Finally, as an entrepreneur in a regulated industry, you may face civil and criminal penalties if the regulator's interpretation of the law holds up, even if you had no awareness or involvement in the issue at hand (1, 2)

Collectively, this may give some insight into why many entrepreneurs have in the past pursued social gaming and photosharing rather than enterprises in regulated industries. Given the totality of evidence, let's postulate that while a lack of imagination may play a role, at least some of the reticence towards doing businesses with big ideas is indeed regulation.

What about the Good Aspects of Regulation?

We are expressly *not* making any policy recommendations in this document; our purpose is to provide a set of facts and articles for the entrepreneur or early engineer to understand many of the details of starting a company in a regulated industry. Because the practical aspects of operating amidst regulation are not covered in most engineering or business school curricula, in order to discuss them we will necessarily present facts that are not commonly circulated, and may thus be at variance with the popular understanding of how regulation is "supposed" to work.

But to engage the point briefly, are there regulations written with good intentions, regulations to manage externalities, regulations to prevent terrorism? Surely there are. And in theory, it may even be the case that the current set of regulations optimally balances Type I and Type II error rates in all circumstances, e.g. that the number of excess deaths due to delayed pharmaceuticals is exactly equal to the number of excess deaths from defective pharmaceuticals. However, in practice this is an empirical question, and one that can be personally adjudicated after you become directly familiar with the details of navigating regulations, preferably after starting a company. Most non-founders are insulated from such questions and lack direct experience with the issues. As a useful analogy, consider the following statements about the Transportation Security Administration (TSA):

1. Millions of people go through TSA checkpoints every day.
2. The regulations are nontechnical and characterized by many as impotent; two permitted three-ounce bottles, for example, may be combined into a banned six-ounce putative weapon-of-mass-destruction through the advanced terrorist technology known as "mixing".
3. The economic stakes associated with missing a flight and/or a meeting at the destination are in the hundreds to thousands of dollars, so people generally nevertheless comply with arguably irrational rules and refrain from making jokes.

4. A traveller is only under the jurisdiction of the TSA for a time-limited interval of a few hours.
5. In theory, then, it should be feasible for travellers to challenge the TSA. In practice, the TSA's [budget](#) has increased from \$1.3B in 2002 to \$8.1B in 2011, and it has been ramping up the [Visible Intermodal Prevention and Response](#) (VIPR) program in its [plan](#) to expand beyond airports to search travellers at train stations, bus stops, subways, and more.

These statements about the TSA should not prove too controversial to anyone who has traveled on a commercial airliner within the US. However, now imagine that the only information you had about the TSA came from its [blog](#) and numerous [social media accounts](#). In the absence of direct experience with a TSA checkpoint, you might then be more likely to believe that what Bruce Schneier has called [security theater](#) has indeed made people safer. Now let's consider founding a company that deals with another regulatory agency, like the SEC.

1. At most a few thousand entrepreneurs contend with the agency each year.
2. The regulations are highly technical and often phrased in a forbidding way; for example, [dihydrogen monoxide](#) may cause severe burns and is fatal if inhaled.
3. The economic stakes associated with being penalized by regulators can range into the [hundreds of millions](#) or [billions](#) of dollars, costing you years of work and your [very freedom](#).
4. You are under the jurisdiction of the regulator for the lifetime of your company, whether you know this at the outset or not.
5. The budget of your regulator is likely to increase over time ([historical data](#), see page 91).

So: there are far fewer people passing through the regulatory aperture, the stakes are in the millions of dollars, the timeframe is years rather than hours, and the penalties for speaking up are significantly greater than making a joke in a TSA line. We are now no longer talking about the possibility of a [retaliatory wait time](#) that makes you miss your flight, but rather the possibility of a [retaliatory denial](#) that ends your company as a going concern. Now, it might nevertheless be the case that your particular regulatory agency is in fact benevolent; we are by no means gainsaying that as a possibility. However, as an entrepreneur, the moment you start a company and walk through the metaphorical metal detectors into a regulated industry, you will be able to judge for yourself whether your regulator generally achieves its stated objectives or whether its activities are best compared to safety theater.

Anticipate the Argument

Whatever the justness of regulation in the abstract you should plan for your specific business, once successful, to become characterized as a threat to health or safety or both. If in doubt, think deeply about how your business could become a tool for terrorism and/or used against the children. For example:

- Uber carries terrorists ([Link](#): *Later, another witness warned that Uber's cars are the perfect weapon for terrorists.*)

- Myspace allows predators to stalk children ([Link](#): *MySpace: Your Kids' Danger?*)
- Paypal can be used to funnel money to terrorists ([Link](#): *[T]he company now has six months. ... to protect its online payment system from being used by money launderers and financers of terrorists.*)
- Google allows children to access inappropriate sites ([Link](#): *Google SafeSearch 'Moderate' Setting Fails to Filter 'Playboy'*)
- Bitcoin is used for money-laundering and terrorism ([Link](#): *Bitcoin might logically attract money launderers, human traffickers, terrorists*)
- Facebook is unsafe for children ([Link](#): *EU: Children unsafe on Facebook*)

If you can't think of something involving terrorists or children, become more creative:

- A personal genomic test can make someone jump off a bridge ([Link](#): *There's been an apocryphal amount of fear in the field that patients may not be ready for this information, that they'll jump off a bridge or whatever if they have a certain gene*)
- Cell phone competition endangers public safety ([Link](#): *Nextel Communications Inc. agreed to spend \$2.8 billion and give up airwaves worth \$2.06 billion in exchange for new spectrum under a U.S. government plan to lessen interference of public safety radios.*)

There will always be some sense in which your product is a threat to health and safety. Everything from [Buckyballs](#) to [spinach](#) can and will be banned/regulated if there is incidence of a health hazard; the actual frequencies of said hazards are less material than the associated press coverage. In general, if you can anticipate the likely line of attack, you may disable it pre-emptively by talking about your various anti-terrorist and/or pro-children features.

The A/B/C/D Theory of Regulation

Let's now step back for a second and see if we can enumerate some general patterns. The fundamental concept behind a *regulation* is as follows: A and B wish to engage in an exchange, but a politically powerful party C finds some aspect of this exchange distasteful and gets party D (the government) to ban it (Figure 4). For example, a store (A) may wish to sell to customers (B) for 24 hours per day. However, the neighbors of this store (C) complain about the late night noise to the local govt (D), who can then choose to impose zoning restrictions that ban commercial transactions between certain hours (example). As another example, a working individual (A) may wish to invest in a company (B), but enough other investors have lost their money (C) that the SEC (D) can choose to ban transactions between A and B. As a third example, an individual (A) wishes to try an experimental drug from a pharma co (B), but others (C) feel that he might hurt himself, and so get the FDA (D) to block this transaction.

Note the term *choose*. Regulatory agencies and local governments are run by humans with their own incentive structures, and they have discretion in choosing whether or not to enforce a law (or issue a new draft guidance, guidance, rule, regulation, or NPRM filing). For example, shift workers might complain to the local government about their inability to obtain supplies during off hours due to the ban on late-night commerce. Or people below the SEC's accredited investor threshold might reasonably demand the freedom to invest in their friend's

The A/B/C/D of Regulation

A and B wish to engage in a transaction. Some feature of this transaction alarms C, a politically powerful actor. D, the government, is then called in to block the transaction.

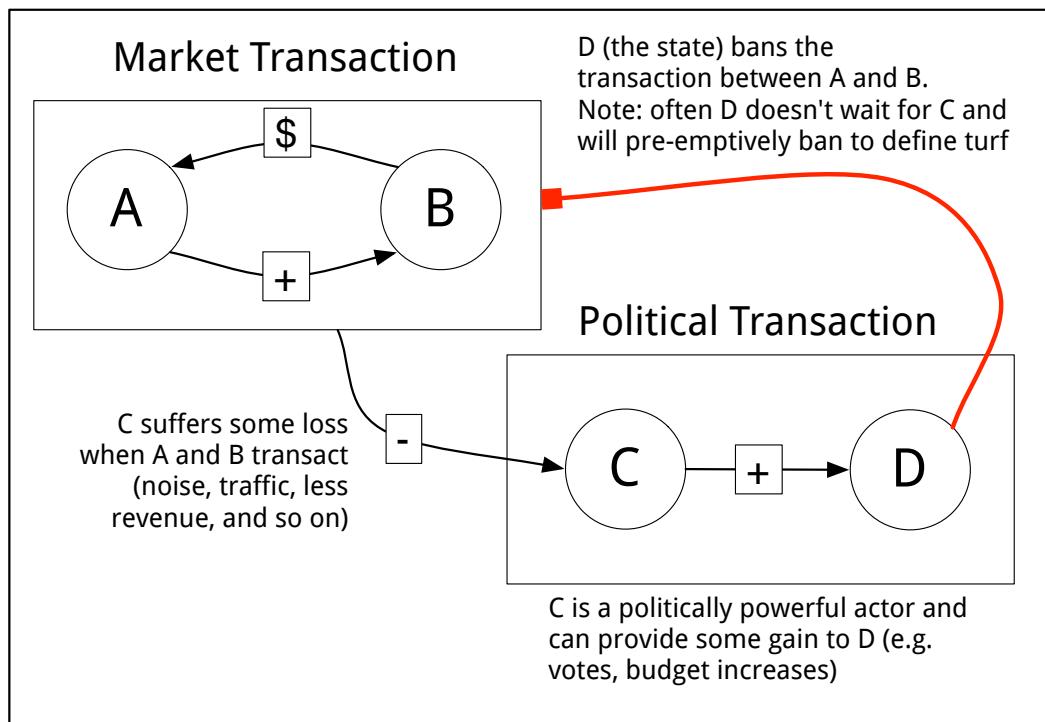


Figure 4: The A/B/C/D Theory of Regulation.

new startup. Which voices then are heard? According to [public choice theory](#), the decision as to which citizen voices are amplified and which are damped often reduces to the question of which voices provide grounds for agency budget/power increases. The reason is simple: like a corporation, those agencies over time that do not grow are absorbed or sidelined by those that do.

Technological Legalization

Importantly, with new technology it is now often feasible to revisit this A/B/C/D relationship and restructure it in some way such that it is legalized. For example, you can set it up such that:

- The A/B transaction is placed into a legal gray area by new technology, and then grows at internet speed such that it becomes “unbannable” (C’s objections are overwhelmed and D lacks the political capital to ban it)
- A or B are organized via the internet to put pressure on D, balancing C’s pressure
- C is no longer vexed by the A/B transaction externality
- C now receives some benefit when A and B transact

- D (the government) receives enough benefit that C's entreaties suddenly fall on deaf ears
- D+ (a superior government agency) receives enough benefit to veto D
- E (a foreign government) receives a large enough benefit that D's disapproval no longer matters and E legalizes the transaction on its terrain.

Many successful companies use one or more of these techniques. Let's go through several case studies in Figure 5.

A (Seller)	B (Buyer)	Transaction	Objection	C (Blocker)	D (Govt)	Startup/Tech Restructuring	Concept
Small business	Customer	24 hour sales	Late night noise	Neighbor	Local gov	Amazon.com open 24 hours	C objection obviated
Payment company	Vendor	Direct sales	Unlicensed money transmitter	Banks and competitors	State gov	Square fundraising for politicians	D+ rev share
Car maker	Car buyer	Direct sales	Unlicensed dealer	Car dealers	State gov	Tesla direct showrooms	D+ support
Private company	Investor	Buy shares	Non-public sale	Bank competitors	SEC	FB/Goldman SIV: non-US	D denied jurisdiction
Citizen driver	Traveller	Hail a cab	Unlicensed taxicab	Taxi unions	Local/State gov	Uber internet petition	A/B org via web vs. D
Startup	Investor	Buy shares	Investor may lose money	Legacy banks	SEC	Angel List online petition	A/B petition of D+
Citizen host	Traveller	Sublet a room	Illegal hotels	Legacy hotels	State gov	Airbnb online booking	D+ support
Car maker	Customer	Install video dashboard	Distracted driving	Other drivers	NTSB	Google self-driving cars	C objection obviated
Plane inventor	Test pilot	Test new plane	Possible harm to pilot	Concerned citizens; FAA	FAA	Unmanned drones	C objection obviated

Figure 5: A regulated transaction in which A is banned from selling to B can be restructured through technology, startup chutzpah, or both such that it becomes legal. See text for more.

Amazon.com: invalidate zoning via web ordering, anesthetizing C

An underappreciated aspect of Amazon.com and all e-commerce sites is that physical zoning restrictions do not apply; aside from the IRS' [EIN site](#), it is rare to find a website that operates only during business hours. Thus, they can use technology to remain open 24 hours per day, placing a storefront on every desk, phone, and lap. One can order grand pianos without making a sound, thereby obviating C's objection of late night noise.

Square: cut in D on the deal via fundraising

As another example, consider the Square dongle. From the financial industry's perspective, this is as disruptive as it gets. All kinds of small merchants can now accept physical credit card payments without paying¹ large banks. And no doubt this is illegal under some statute, or can be [deemed so](#), especially if there is enough pressure from existing financial institutions. But Square has canny executives, and they spent quite a lot of effort on the Square fundraising app ([1](#), [2](#), [3](#)). One can now envision the situation in which a lobbyist for BigBank is talking to a politician about banning Square, clearly a crime against humanity, decency, and the American way. An aide whispers in said politician's ear ("Sir, we raised \$100,000 over the last three months using Square") and now the bank's entreaty falls on deaf ears. In this manner one can co-opt D before C does.

Tesla: cut in D+ on the deal to overcome local car dealers

Tesla received a large [\\$465M loan](#) from the Department of Energy. While many of the other companies in that program (A123, Solyndra, etc.) have ended up failing or going [bankrupt](#), Tesla is developing into a success. As such the federal government (D+) has a strong incentive to remove unnecessary roadblocks from Tesla's path. In particular, now that local car dealers have started objecting that direct sales violated [state law](#), Tesla has the political connections to bring in [federal favors](#) to overwhelm the state governments ("Tesla may take dealer fight to feds"), should that be necessary.

Facebook/Goldman: Market to non-US individuals, denying D jurisdiction

In 2011, the New York Times ran a [negative article](#) on an upcoming investment in Facebook by various private investors, with the round managed by Goldman Sachs. The SEC reprioritized based on the front page NYT article and began [sending letters](#) to Facebook and Goldman, indicating that to continue with the transaction would be illegal unless they filed for an IPO. Since becoming a public company is an extraordinarily [complex](#) and lengthy process, this would have put the kibosh on the investment. Had the capital been needed for servers and salaries and the like, it might even have put the kibosh on Facebook itself. So [Facebook's solution](#) was simply to offer the transaction to non-US investors, thereby obviating the SEC's claim of jurisdiction. This option of *jurisdictional arbitrage* is a relatively new phenomenon that has been made feasible by the economic rise of the BRICs and the Asian Tigers relative to the West.

Uber: Use the internet to organize A/B against C/D

Uber is one of the companies that is in the thick of it with respect to regulation (Figure 6). Their initial [smartphone app](#) for internet booking took a loophole in the law and drove a truck (or a black car) through it. This allowed them to grow rapidly in the early days before the

¹Square is another example of a good idea where there is knowledge of a term that people outside the industry don't know. At first one might think Square was a bad idea, as you'd think people could just type in their card details into a mobile phone or tablet, rather than going to the trouble of actually building and installing a physical hardware device. At least one of the underlying reasons is the distinction between CP (card present) vs. CNP (card not present). A transaction that includes a physical card swipe is labeled CP: because the card is present, there is fundamentally lower risk of fraud and hence higher profit margins for Square.

law caught up to them. Crucially, once the [cease and desist](#) letters started flying, Uber only faced state/local regulation rather than federal regulation. As such they could monetize in one city while fighting regulations in another city. This is very different² from dealing with a federal regulator like the SEC; to avoid a federal regulator requires going outside the country (as in the case of Facebook and Goldman).

²Moreover, travellers like James Fallows could use Uber in one city while on the road and then be dismayed when it was banned in their home city. To do this with a federally regulated product would require travelling outside the country, which is less common.

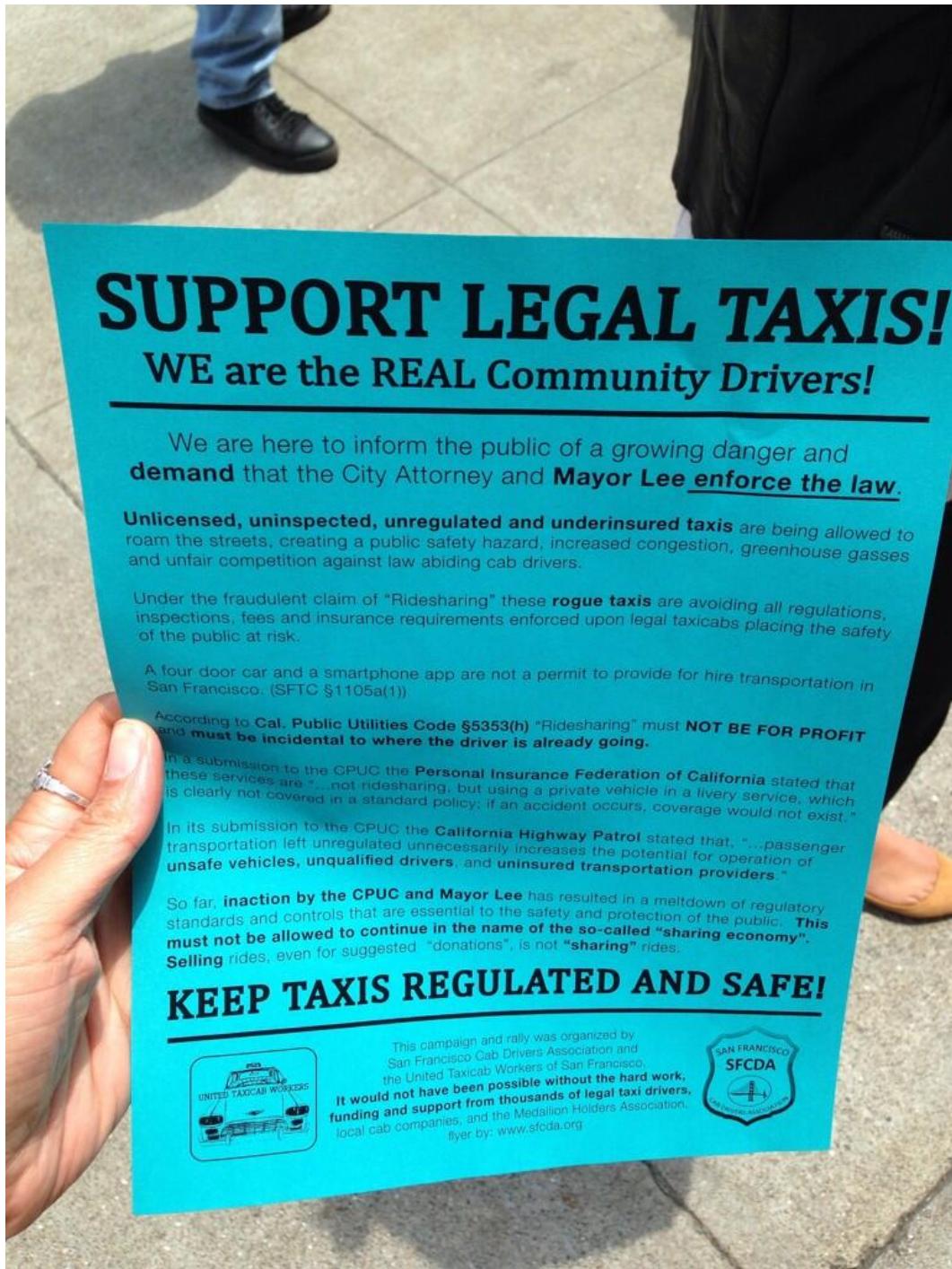


Figure 6: A flyer calling for Uber to be banned due to "unfair competition". [Source](#).

And this led to a second Uber tactic for dealing with regulation: use the internet's capacity for direct democracy to organize A and B against C (competing cab drivers) and D (local government). Local politicians like [Mary Cheh](#) who'd enjoyed cozy relationships with the local taxicab industry suddenly found that they had misestimated the balance [of forces](#):

In many of those battles, Uber's secret weapon has been its customers: The kind of well-heeled, tech-savvy urbanites willing to pay a hefty markup to avoid the annoyance of hailing a cab. They may never before have shown an interest in any other aspect of local governance. But when some taxi commissioner or city councilor tries to take away their newfound convenience, they'll rally to its defense with calls, e-mails, and indignant tweets. . . . "We are building a playbook for how to do this," [Uber CEO Kalanick] told a Washington, D.C., audience of policy wonks in January. "Other companies are going to follow suit in all kinds of industries that each is affecting. And I think folks in D.C. and cities across the country, you're going to be in the middle of it."

Perhaps the most important quality that Uber has is the psychological unwillingness to submit, the defiance to actually fight. Because with venture capital and superior technology, it is for the first time in recent memory possible to [fight city hall](#), and win.

Angel List: Use A/B to petition D+ and change D's jurisdiction

In the middle of 2012 something unusual happened. The JOBS Act included a provision that actually legalized equity-based crowdfunding, unbanning certain kinds of transactions. As [Ben Horowitz](#) of A16Z said:

When Ben Horowitz stopped by PandoMonthly last June, he celebrated the work done by AngelList's Naval Ravikant to help push the Jumpstart Our Business Startups (JOBS) Act through Congress. Horowitz said, "He went to Washington and he changed the law! That blew me away."

. . . Ravikant set out to convince lawmakers that the JOBS Act, which eases securities laws, would encourage more small business investment. Any one who's seen a debate or political speech over the past four years knows that "building small businesses" is golden rhetoric for politicians on both sides of the aisle. But even amid this climate, it wasn't easy. "People told us that it was impossible - and it actually basically is impossible," Ravikant said. "We just pulled out all the stops."

This provision did not come without quite a fight. As context, Angel List, Funders Club, and several other crowdfunding platforms were living for many years with the Sword of Damocles over their heads. At any time, the SEC's David Blass could decide that they were in violation of the law and choose to press charges against them (they eventually chose [not to](#)). This is known as [enforcement discretion](#). In the DOJ, for example, US Attorneys like Carmen Ortiz have [plenary authority](#) in their territory and the ability to [press charges or not](#) as is their wont (see also [Three Felonies a Day](#)). Any civilian will be familiar with this in the context of the highway patrol: a policeman has the power to pull you over, and need not justify his decision to *not* pull all the *other* speeding motorists over.

What Angel List did was phenomenal: even with this Sword of Damocles hanging over them, they coordinated a [campaign of signatures](#) in favor of the new provisions in the JOBS Act. The entrepreneurs (A) and investors (B) who benefited from Angel List were able to organize through the internet to outweigh the interest groups (C) and regulators (D).

And More...

At this point you may begin to get the idea of how one can legalize previously banned transactions through a combination of technology and business strategy. Airbnb used their online booking system to host people during [Hurricane Sandy](#) and put out a joint press release with Michael Bloomberg at [nyc.gov](#); this support from D+ (the Mayor) may help in their [new battle](#) with the city itself. The Department of Transportation imposed [many restrictions](#) on the kinds of devices that can be included in cars, with the idea being to avoid driver distraction; but if self-driving cars are legalized this issue will be obviated (though this itself is [in doubt](#)). And, as a third example, there are stringent rules on test pilots that must be followed for new aircraft, but with [unmanned aerial vehicles](#) (i.e. drones) one can in theory fail many more times with more aggressive designs before including the first human pilot. In short, there is often a way to use technology to restructure a banned transaction such that it is over the threshold of acceptability.

Disruption and the Technologies of 2013

Let's talk now for a bit about what it means for software to eat the world, in the context of some of the most important³ technologies of 2013. We begin with six technologies and premises about what the coming decades will be like.

1. Management is Automation (Industrial Robotics)
2. Regulation is DRM (3D Printing)
3. Immigration Policy is a Firewall (Telepresence)
4. Medicine is Mobile (Quantified Self, Telemedicine)
5. Capital Controls are Packet Filtering (Bitcoin)
6. Warfare is Software (Autonomous Drones)

Let's elaborate on these statements.

Industrial Robotics: Management is Automation

Perhaps the defining industrial innovation of the 20th century was the assembly line. For the 21st, it may be industrial robotics. While factories have had robots of some form for quite a while, recent advances in robotics have made it possible to write code to automate whole facilities ([1](#), [2](#), [3](#), [4](#)). Even tasks that require extreme manual dexterity will soon be done by [robot hands](#).

³We intentionally exclude photosharing and social gaming for now, as well as most ecommerce and enterprise app plays, as most of the difficulty in these companies is in the execution rather than the technology itself. There's nothing wrong with that, it's just not as interesting from a technology perspective.

This will have significant consequences. Among other things, by turning management into code, management now becomes tangible. As context, the stereotypical image of the manager in the early 20th century was a fatcat boss with his feet up on a desk smoking a cigar, while workers slaved away below on the assembly line. From today's vantage point, however, we can start to assign some value to the employee training manuals, [assembly line layouts](#), and even internal memos that those 20th century managers used to organize their factories. This is because the 21st century manager of an assembly line can replace employee training with scriptable machine images, can substitute architectural diagrams for assembly line configurations, and can think in terms of message passing protocols rather than internal memos. In other words, the previous intangible informational tasks done by the manager now become tangibly recorded in git logs and database entries. In this world the manager blends into the worker, and the assembly line morphs into a robotic factory scripted by said manager/worker.

Not only does this mean far fewer workers, it means far fewer constraints. For better or for worse, no employment law provisions apply to robots. There are no hourly restrictions, minimum wage laws, collective bargaining agreements, or decommissioning restrictions ([WARN act](#)) when it comes to robots. [OSHA](#)'s power over the workplace also plummets when there are no workers. In many ways this scenario is alarming to not just the already embattled US factory worker, but to workers at Chinese companies like Foxconn that may be [replaced](#) by robots. The flipside, though, is that as this technological trend begins to accelerate, the capital requirements to mass produce a good will decline precipitously. To run a small robotic factory will be like running one's own datacenter, well within reach of the individual entrepreneur.

3D Printing: Regulation Becomes DRM

3D printing ([1](#), [2](#), [3](#)) has been [hailed](#) as the replacement for the assembly line, as a tool that will allow at-home manufacturing. However, it's hard to see how the first generation of 3D printers can produce plastic goods that have the quality and robustness of highly-optimized, mass produced metal items. One could make the argument that 3D printing can enable just-in-time manufacturing and reduce supply chain lag times and inventory volumes, and that likely will be true as the field progresses, but that's not likely to be the most immediate application if the plastic goods produced are significantly below the quality of mass-produced goodsa.

An alternative view is that the most important consequence of 3D printing will be to make it impossible to ban goods. Already we have working versions of 3D printed [guns](#) ([1](#), [2](#), [3](#), [4](#), [5](#), [6](#)), [drones](#), [cars](#), and [medical devices](#). And in the UK Professor Lee Cronin is working on a way to 3D print pharmaceuticals and other chemicals ([1](#), [2](#), [3](#)). People have not yet begun to process the consequences of this. A whole alphabet soup of agencies, from the ATF to the DHS to the FAA to the FDA to the NHTSB derive their raison d'etre from banning or limiting access to these controlled goods. But now, because information can be freely transmitted across the internet and then reconstituted into a physical object via a 3D printer it will be impossible to ban these goods. As Homeland Security [itself notes](#):

"Significant advances in three-dimensional (3D) printing capabilities, availability of free digital 3D printer files for firearms components, and difficulty regulating file sharing may present public safety risks from unqualified gun seekers who obtain or manufacture 3D printed guns," reads a May 21 bulletin . . . "Limiting access may be impossible."

Still, they exist on the same peer-to-peer file sharing services that distribute pirated entertainment (and legal software). “Even if the practice is prohibited by new legislation, online distribution of these digital files will be as difficult to control as any other illegally traded music, movie or software files.”

And consider in particular these four points:

1. There are now one billion internet-connected, unlockable Android devices built on a Linux kernel
2. VPNs can be used to [defeat](#) deep packet inspection
3. [Steganography](#) can be used to hide substantive payloads in the photos and videos sent over social networks
4. It has generally proven infeasible to [ban encryption](#) or even [3D printing](#) due to their substantial non-infringing uses

Taken together and projected out a few years, this means it is now possible to envision a world aflood with tiny \$35 [Raspberry Pi](#) servers chock full of banned goods. If history is any indicator, the alphabet soup will not completely give up without a fight. It will seek to push the hardware manufacturers to install DRM on their printers to disallow the printing of particular form factors (and this has [already begun](#)). This will be similar to the efforts to disable Photoshop from [editing dollar bills](#), or the various efforts of Hollywood to put [digital rights management](#) into media players. In other words, the ability of regulators to control access to prohibited devices will reduce to the strength of their DRM.

But because we are dealing with continuous objects rather than discrete ones, the ability of DRM to actually block the printing of certain forms is likely to be a cat and mouse game. For example, consider the evolution in just two months time from a [single shot 3D printed gun](#) to a [multishot gun](#) to a full [3D printed rifle](#) capable of firing fourteen rounds. Even without fully disabling DRM, one would only need to continuously deform an object enough to 3D print it.

Telepresence: Immigration Policy is a Firewall

Another interesting technological trend is the rapid improvement in telepresence. Beginning with Trevor Blackwell’s [Anybots](#), a host of new companies like [Double Robotics](#) have arisen that provide a video-game like interface to a remote machine. Rather than using the [W/A/S/D](#) keys to navigate through the mazes of Doom, you now use them to remotely attend a happy hour in San Francisco while you are in Bangalore. This is telepresence: not simply video chat, but video chat on wheels.

The technology is just getting started, but project it out a few years. Combine [Oculus Rift](#), [Myo](#), [3D Treadmill](#) as input and [Double Robotics](#), [Petman](#), and [Google Glass](#) as the remote machine. You now have a device similar to that from the movie [Surrogates](#). You would be using the three input devices to remote control a humanoid robot in another country or jurisdiction. Said robot would be able to record and stream its environment back to you. Though the system integration will be nontrivial, all six of these technologies exist today and are individually functional.

The whole will be greater than the sum of its parts, however. Telepresence of this kind will become an imperfect but extremely cheap substitute for a tourism junket or work visa.

And because physical presence is no longer necessary, nation states will find that the only way to restrict access to certain immigrants will be to set up a [firewall](#) that restricts particular kinds of international telepresence connections. In this manner, a nation's immigration policy will be as good as its firewall.

Quantified Self: Medicine is Mobile

We spoke about the quantified self trend the last time, but it's worth thinking about this a bit more. The combination of [telemedicine](#), [mobile appointment booking](#), [quantified self](#), 3D printed [devices](#) and [drugs](#), and [medical tourism](#) will make it much more difficult to limit access to new biomedical technologies. Put these together and you will find people routinely self-diagnosing via quantified self, doing a video chat with a physician in another state or country, traveling to meet that physician for surgery, or even 3D printing their own medical devices or drugs.

The last step might seem particularly radical, but keep two points in mind. First, the nature of open source software platforms is that over time even this process of drug printing or device printing will be turned into a one-click operation. It's very hard to write a video codec or to produce a feature film, but it's not that hard to download a torrent and watch it on [VLC](#). In the same way, the open source community will quickly refine device and drug plans and circulate them for use by early (and then not-so-early) adopters. And some of these open source contributors will even be [skilled physicians](#).

Second, if a patent or a paper on a drug is publicly available and yet someone can't get access to it for monetary or regulatory reasons, they may choose to try a radical step rather than simply surrender to their illness. In Cowan vs. US, for example, a terminal patient [sued](#) to get access to a drug that was being kept from him by [federal regulations](#).

Plaintiff and Plaintiff's doctor testified that Plaintiff is terminally ill. . . .

The Court is sympathetic to Plaintiff's situation. However, the law is very clear, and under the current statutes and regulations, Plaintiff's physician may not administer the goat neutralizing antibody drug absent prior approval of the FDA. In Court, Plaintiff argued that he should have the right to take whatever treatment he wishes due to his terminal condition regardless of whether the FDA approves the treatment as effective or safe, and that to prohibit him from taking the treatment he wishes violates his rights under the United States Constitution.

. . .

This Court is in no way criticizing the intentions of Plaintiff and his physician or the potential effectiveness of the proposed treatment. Plaintiff's physician should pursue approval of his Investigational New Drug application as quickly as possible. Plaintiff's doctor must obtain appropriate approval through the proper regulatory authorities. As much as this Court may empathize with Plaintiff, the authority to provide some type of exemptions for individuals such as Plaintiff rests with Congress and not with this Court.

It's difficult to read about what happened to Mr. Cowan; he's probably passed on now. Someone in a similar situation in 2018 or 2023 might simply (a) 3D print their drug or (b) find a prescribing physician via telemedicine, travel overseas, and take his drug in another country. In this manner medicine may become mobile.

Bitcoin: Capital Controls are Packet Filtering

Bitcoin is a new kind of digital currency, native to the Internet, that can be variously thought of as an open-source project (like Linux), a protocol (like HTTP), or a commodity (like gold); see this [short introduction](#) for more. The fundamental innovation behind Bitcoin is a breakthrough in [distributed systems](#); the underlying Bitcoin protocol can now be used to distribute many kinds of algorithms that were thought to involve a central server, including transaction processing. Specifically, rather than increment or decrement an account's balance on a central bank server, Bitcoin has a [clever way](#) of recording it on a distributed network of computers and updating it as you send and receive payments.

Though the distributed Bitcoin network [began](#) processing transactions all the way back in January 2009, the Cyprus bank account seizures of March 2013 were by [many accounts](#) a major shot in the arm for Bitcoin adoption. Bitcoin prices spiked all the way up to \$266 USD/BTC before falling back down to a relatively stable \$100 USD/BTC. Cypriot bank accounts were frozen and funds were seized, with the final toll being [at least](#) 47.5% of assets on all accounts with at least \$132,000 in savings:

Depositors at bailed-out Cyprus' largest bank will lose 47.5% of their savings exceeding 100,000 euros (\$132,000), the government said Monday.

The figure comes four months after Cyprus agreed on a 23 billion-euro (\$30.5 billion) rescue package with its euro partners and the International Monetary Fund. In exchange for a 10 billion euro loan, deposits worth more than the insured limit of 100,000 euros at the Bank of Cyprus and smaller lender Laiki were raided in a so-called bail-in to prop up the country's teetering banking sector.

The savings raid prompted Cypriot authorities to impose restrictions on money withdrawals and transfers for all banks to head off a run. Christopher Pissarides, the Nobel laureate who heads the government's economic advisory body, forecast Monday that the bank controls could be in place for another two years.

More bailouts and haircuts for Cypriots [may follow](#):

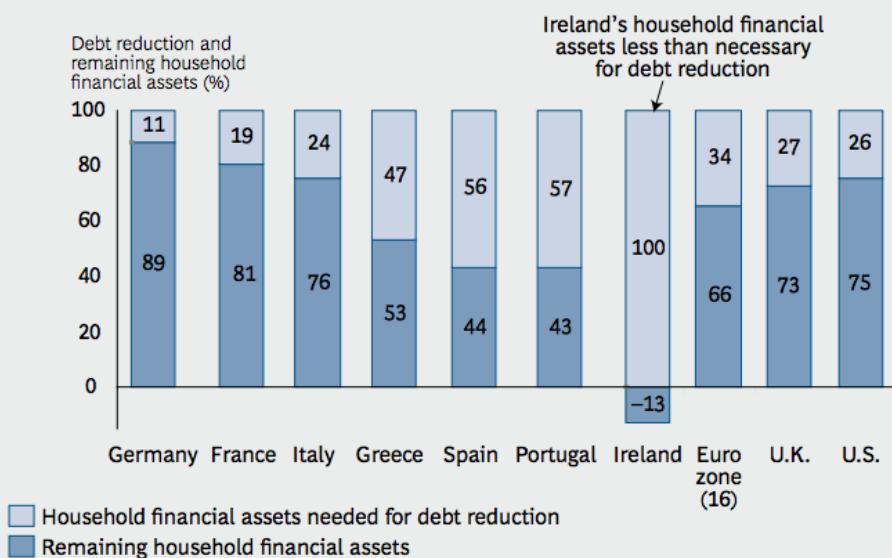
The Cyprus government is looking to the European Central Bank to provide a restructured Bank of Cyprus with as much liquidity as it needs to help turn the country's tanking economy around by lending to cash-starved businesses. Anastasiades last month warned ECB chief Mario Draghi that Bank of Cyprus' cash reserves were running dangerously low.

And for the last several months, stringent [capital controls](#) have been imposed on Cyprus, limiting the amount of money that can be taken out of the country. Finally, well before the Cyprus haircuts occurred BCG published a [document](#) that calculated the size of haircuts required in 10 countries with high debt to GDP ratios (Figure 7).

EXHIBIT 3 | One-Time Wealth Tax That Would Be Required to Cover the Costs of Debt Restructuring

Necessary debt reduction to reach 180 percent debt-to-GDP ratio

€ (billions) 523 727 845 134 998 221 340 6,121 1,252 8,243



Sources: Eurostat; Federal Reserve; Thomson Reuters Datastream; BCG analysis.

Note: All data as of 2009.

Figure 7: BCG's estimated wealth tax in ten industrialized nations, as of 2011. ([Source](#))

While it certainly will not happen overnight, if Bitcoin grows in adoption it will make wealth seizures and capital controls much less feasible. Wealth seizures become much more difficult because Bitcoin [private keys](#) can be held on local laptops, stored on USB keys, or printed on paper wallets. Unlike bank accounts held on the servers of private banks, these air-gapped Bitcoin storage systems are not detectable or automatically debitible by central systems.

More interestingly, capital controls also become less feasible. Traditionally one has to declare at customs if one is carrying large amounts of physical cash ([1](#), [2](#), [3](#)). But Bitcoin defeats this paradigm of capital controls. Not only can Bitcoin be transferred through the internet, the private keys to a Bitcoin address can be put into a cloud folder. One could then imagine a traveller who steps through an airport and truthfully declares that s/he has no cash on their person. Then, once on the other side, said traveller simply downloads their private keys from their Dropbox or Google Docs folder, or their own personal server which is still physically present in Peoria, Kansas. And truth be told, such a trip is not even genuinely necessary. So long as an encrypted connection can be established between two parties, Bitcoin can be sent between those parties. And if [Zerocoin](#) and the like continue to improve, the ability to trace that transaction through the [Bitcoin Blockchain](#) will also grow cold. In other words, we may be approaching a future where the ability to control the entrance and exit of capital from a jurisdiction reduces to the ability to perform [deep packet inspection](#) and [packet filtering](#), searching for signatures of Bitcoin transactions.

Autonomous Drones: Warfare is Software

Finally, let's talk about drones. [Self-driving cars](#), [walking robots](#), and [unmanned aerial vehicles](#) are about allied technologies: the ability for a machine to navigate from point A to point B without constant human intervention and correction. Until you've actually flown a drone like the consumer Parrot AR machines, it's easy to forget the fundamental difference between drones and RC planes: unlike RC planes, drones can fly outside the line of sight without constant ground control.

This is somewhat magical. It means that you can, in theory, set waypoints for a drone and then sit back and have it [deliver you a taco](#), steering past obstacles and landing with aplomb. But the applications move beyond tacos. Right now it requires [five soldiers](#) to control a single Predator drone. But over time the number of drones controller per individual operator (the fan out) will rise from 1:5 to 5:1 and beyond. And those drones will be not just aircraft but humanoid drones.

As that number rises, military strategy starts to resemble a real-life game of Starcraft. Everything becomes about controlling your drones. And then everything becomes about finding [zero-day](#) vulnerabilities in the drones and/or drone control systems of your opponent. By taking control of their drones you can turn them against the enemy or set them to self-destruct. Ultimately what this means is that warfare becomes more than ever about quality, not quantity. The number of soldiers won't matter; to win a war, you will need better software.

Coda

When we put these technological trends together we can see a future of radical, disruptive legalization through technology. Everything of significance is reduced to whether or not individuals can securely transmit packets to each other, via VPN or steganography or other means. If that right is preserved, it will be impossible to block these technologies. And with

close to 1 billion mobile internet connected devices released in the world, it can be argued that the cat is already out of the bag. Unlocking those mobile devices and using them to send/receive encrypted packets over the internet is likely to be the means by which this future is achieved.

There is, however, one remaining potential roadblock: the actual physical internet backbone itself. International transmission of information still travels over a huge worldwide network of underseas cables, and in the event of serious exigency these cables are still under the control⁴ of traditional nation states. Google's [Project Loon](#) is perhaps the most interesting way to address this, with a network of high-flying balloons providing internet connectivity.

Whether Loon will provide an unblockable internet or not remains to be seen, but if everything else becomes virtual and if these technologies continue to advance we can expect the physicality of the internet backbone to move from the background to a battleground. If that does in fact happen - if interference with the physical backbone becomes the only way for nation states to stop disruptive technological legalization, to reassert their jurisdiction via packet filtering, firewalls, and DRM - at least one thing is certain. No one will contend any more that Silicon Valley's imagination is limited to social gaming and photosharing.

⁴When Syria [stopped all internet traffic](#) leading into the country for a period of 48 hours, the initial speculation was that the government had actually cut the cables, but eventually it looked like they had just gone after the [routing tables](#). See Cloudflare ([1](#), [2](#), [3](#)) for more details.

Node.js: Asynchrony, Flow Control, and Debugging

In this lecture we will discuss latency, asynchronous programming with the `async` flow control library, and basic debugging with the [Node debugger](#). To begin, it's useful to understand at a high level what problems Node is trying to solve, what it's good for, and what it's not suited for.

Motivation: reduce the impact of I/O latency with asynchronous calls

To understand the motivation behind Node's systematic use of asynchronous functions, you need to understand latency. The first thing to know is that different computer operations can take radically different amounts of time to complete, with input/output (I/O) actions such as writing to disk or sending/receiving network packets being particularly slow (Table 1).

Table 1: Latencies associated with various computer operations as of mid-2012. Here, 1 ns is one nanosecond (10^{-9} s), and 1 ms is one millisecond (10^{-3} s). (Source: [Jeff Dean](#) and [Peter Norvig](#).)

Operation	Time (ns)	Time (ms)	Notes
L1 cache reference	0.5 ns		
Branch mispredict	5 ns		
L2 cache reference	7 ns		14x L1 cache
Mutex lock/unlock	25 ns		
Main memory reference	100 ns		20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns		
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms	
Read 4K randomly from SSD*	150,000 ns	0.15 ms	
Read 1 MB sequentially from memory	250,000 ns	0.25 ms	
Round trip within same datacenter	500,000 ns	0.5 ms	
Read 1 MB sequentially from SSD*	1,000,000 ns	1 ms	4X memory
Disk seek	10,000,000 ns	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20 ms	80x memory, 20X SSD
Send packet CA -> Netherlands -> CA	150,000,000 ns	150 ms	

To illustrate why these latency numbers are important, consider the following simple example, which downloads a number of URLs to disk and then summarizes them:

```
1 var results = [];
2 for(var i = 0; i < n; i++) {
3     results.push(download(urls[i]));
4 }
5 summarize(results);
```

In this code we wait for each download to complete before we begin the next one. In other words, the downloads are *synchronous*. This means that each line of code executes and completes in order. If we conceptually unrolled the for loop it would look like this:

```

1 var results = [];
2 results.push(download(urls[0])); // block till urls[0] downloaded
3 results.push(download(urls[1])); // block till urls[1] downloaded
4 results.push(download(urls[2])); // block till urls[2] downloaded
5 // ...
6 results.push(download(urls[n])); // block till urls[n] downloaded
7 summarize(results);

```

Each line here completes before the next begins. That is, the script hangs or *blocks* on each line. Now let's define the following mathematical variables to analyze the running time of this simple code block:

- V : time to initialize a variable like `results`
- D : time to download a url via `download`
- A : time to append to an array like `results`
- S : time to calculate `summary(results)`

Then our total running time for the synchronous version (T_s) is:

$$T_s = V + D_1 + A_1 + D_2 + A_2 + \dots + D_n + A_n + S$$

Crucially, from the latency numbers in 1, we can assume that $D_i \gg V$ and $D_i \gg A_i$, though S might be quite long if it's doing some heavy math. Thus we can simplify the expression to:

$$T_s \approx D_1 + D_2 + \dots + D_n + S$$

Now, the reason that D_i takes so long is in part because we are waiting on a remote server's response. What if we did something in the meantime? That is, what if we could enqueue these requests in parallel, such that they executed *asynchronously*? If we did this, we would still start the downloads in order from 1 to n , but they could complete all at different times, and this would be a bit confusing. However, our new asynchronous running time (T_a) would now look like this:

$$T_a = \max(D_1, D_2, \dots, D_n) + S$$

The reason it's the max with the asynchronous implementation is because we are only waiting for the slowest download, which takes the maximum time. And because $D_i > 0$ (as delay times must be positive), this is in theory strictly faster than the synchronous version:

$$\max(D_1, D_2, \dots, D_n) < D_1 + D_2 + \dots + D_n$$

In other words, it could be faster (potentially much faster) if we could kick off a number of asynchronous requests, where we wouldn't *block* on the completion of each download before

beginning the next one. Our wait time would be dominated by the maximum delay, not the sum of delays. This would be very useful if we were writing a [web crawler](#), for example. And this is exactly what Node is optimized for: serving as an orchestrator that kicks off long-running processes (like network requests or disk reads) without blocking on their completion unless necessary. As a concrete example, take a look at [this script](#), which can be run as follows:

```
1 wget https://d396qusza40orc.cloudfront.net/startup/code/synchronous-ex.js
2 npm install async underscore sleep
3 node synchronous-ex.js
```

If we execute this script, you will see output similar to that in Figure 1. As promised by theory, the synchronous code's running time scales with the sum of all individual delays, while the asynchronous code has running time that scales only with the single longest delay (the max).

```
[ubuntu@ip-172-31-28-87:~]$node synchronous-ex.js
Synchronous start at Thu Aug 01 2013 04:05:29 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 start at Thu Aug 01 2013 04:05:29| GMT+0000 (UTC)
  http://www.bing.com/search?q=0 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 elapsed time: 862
  http://www.bing.com/search?q=1 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 elapsed time: 838
  http://www.bing.com/search?q=2 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 elapsed time: 103
  http://www.bing.com/search?q=3 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 elapsed time: 765
  http://www.bing.com/search?q=4 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=4 end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 elapsed time: 214
Sum of times: 2774.613959947601 ms
Max of times: 860.0957898888737 ms
Synchronous end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
Synchronous elapsed time: 2793

Asynchronous start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=1 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=2 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=3 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=4 start at Thu Aug 01 2013 04:05:31| GMT+0000 (UTC)
  http://www.bing.com/search?q=2 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 elapsed time: 101
  http://www.bing.com/search?q=4 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 elapsed time: 213
  http://www.bing.com/search?q=3 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 elapsed time: 763
  http://www.bing.com/search?q=1 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 elapsed time: 838
  http://www.bing.com/search?q=0 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 elapsed time: 860
Sum of times: 2774.613959947601 ms
Max of times: 860.0957898888737 ms
Asynchronous end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
Asynchronous elapsed time: 862
```

Figure 1: The red boxes show that sync code scales with the sum, while async with the max. The blue boxes show the start times of the first and last sync vs. async downloads. The key is that async is non-blocking: each download does not wait for the next to complete before beginning.

We start to see why `async` could be handy. The synchronous code is considerably slower than the equivalent asynchronous code here, because (1) there was no need for each download to *block* on the results of the previous download and (2) there is nothing we could have done within the Node process itself to significantly speed up each individual download to file. The IO bottleneck is on the remote server and the local filesystem, neither of which can be easily sped up (Table 1). Now let's take a look at the source code to see how this example works:

```
1  /*
2   * Illustrative example to compare synchronous and asynchronous code.
3   *
4   * We demonstrate in particular that synchronous code has a running time
5   * that scales with the sum of individual running times during the parallel
6   * section of the code, while the equivalent async code scales with the
7   * maximum (and thus can be much faster).
8   *
9   * We also show examples of object-oriented programming (via Timer) and
10  functional programming (via build_insts).
11
12  To install libraries:
13
14  npm install async underscore sleep
15  */
16 var async = require('async');
17 var uu = require('underscore');
18 var sleep = require('sleep');
19
20
21 /*
22  Illustrates simple object-oriented programming style with JS. No
23  inheritance here or need for prototype definitions; instead, a fairly
24  self explanatory constructor for a simple object with some methods.
25
26  As shown below, we pass new Timer instances into asynchronous code via
27  closures so that they can track timings across the boundaries of async
28  invocations.
29 */
30 var Timer = function(name) {
31     return {
32         name: name,
33         tstart: null,
34         tend: null,
35         dt: null,
36         start: function() {
37             this.tstart = new Date();
38             console.log("%s start at %s", this.name, this.tstart);
39         },
40         end: function() {
```

```

41         this.tend = new Date();
42         console.log("%s end at %s", this.name, this.tend);
43     },
44     elapsed: function() {
45         this.end();
46         this.dt = this.tend.valueOf() - this.tstart.valueOf();
47         console.log("%s elapsed time: %s ms", this.name, this.dt);
48     }
49 };
50 }
51 */
52 Illustrates the functional programming (FP) style.
53
54 We create a set of dummy URLs by using underscore's map and range
55 functions.
56
57 Then we create a set of delays, in milliseconds, to use consistently
58 across both the sync and async implementations.
59
60 Finally, we use underscore's zip function () a few times to help
61 us create a final data structure that looks like this:
62
63
64 [ { url: 'http://www.bing.com/search?q=0',
65   delay_ms: 860.4143052361906 },
66   { url: 'http://www.bing.com/search?q=1',
67   delay_ms: 91.59809700213373 },
68   { url: 'http://www.bing.com/search?q=2',
69   delay_ms: 695.1153050176799 },
70   { url: 'http://www.bing.com/search?q=3',
71   delay_ms: 509.67361335642636 },
72   { url: 'http://www.bing.com/search?q=4',
73   delay_ms: 410.48733284696937 } ]
74
75 The reason we like a list of objects like this is that each individual
76 element can be passed to a single argument function in a map invocation,
77 which can then access the object's fields and do computations on it.
78 */
79 var build_insts = function(nn) {
80     var bingit = function(xx) { return 'http://www.bing.com/search?q=' + xx; };
81     var urls = uu.map(uu.range(nn), bingit);
82     var delays_ms = uu.map(uu.range(nn), function() { return Math.random() * 1000; });
83     var to_inst = function(url_delay_pair) {
84         return uu.object(uu.zip(['url', 'delay_ms'], url_delay_pair));
85     };
86     return uu.map(urls, delays_ms), to_inst;
87 }

```

```

88 /*
89  *
90  * Simple code that uses underscore's reduce to define a sum function.
91  * As we'll see, synchronous code scales with the sum of times while
92  * async code scales with the max.
93  */
94 */
95 var summarize = function(results) {
96     var add = function(aa, bb) { return aa + bb;};
97     var sum = function(arr) { return uu.reduce(arr, add);};
98     console.log("Sum of times: %s ms", sum(results));
99     console.log("Max of times: %s ms", uu.max(results));
100 };
101 /*
102 */
103 /*
104 A straightforward synchronous function that imitates (mocks) the
105 functionality of downloading a URL. We take in an inst object of the
106 form:
107
108     inst = { url: 'http://www.bing.com/search?q=1',
109             delay_ms: 91.59809700213373 }
110
111 For illustrative simplicity, we fake downloading the URL here by simply
112 doing a synchronous sleep with the sleep.usleep function. That is, we
113 halt the process for delay_us seconds. The reason we do a fake download
114 is that this way we can do an apples-to-apples comparison of an async
115 version of the code in which each download took exactly the same time.
116
117 We add two spaces to the beginning of the Timer invocation for
118 formatting purposes in STDOUT, like indenting code.
119 */
120 var synchronous_mock_download = function(inst) {
121     var tm = new Timer('  ' + inst.url);
122     tm.start();
123     var delay_us = inst.delay_ms * 1000;
124     sleep.usleep(delay_us);
125     tm.elapsed();
126     return inst.delay_ms;
127 };
128 /*
129 A straightforward synchronous way to start a time, iterate over a bunch
130 of URLs, download the files to disk, accumulate the times required to
131 download those files, summarize the results and then stop the timer.
132 */
133 var synchronous_example = function(insts) {
134     var tm = new Timer('Synchronous');

```

```

135     tm.start();
136     var results = [];
137     for(var ii = 0; ii < insts.length; ii++) {
138         results.push(synchronous_mock_download(insts[ii]));
139     }
140     summarize(results);
141     tm.elapsed();
142 };
143
144
145 /**
146 Functionally identical to synchronous_example, this version is
147 written to be structurally more similar to asynchronous_example
148 for comparative purposes. Note that the loop is replaced with a
149 map invocation and sent directly to summarize, which is
150 the equivalent of a callback (i.e. directly acting on the results
151 of another function).
152 */
153 var synchronous_example2 = function(insts) {
154     var tm = new Timer('Synchronous');
155     tm.start();
156     summarize(uu.map(insts, synchronous_mock_download));
157     tm.elapsed();
158 };
159
160 /**
161 Like ths synchronous_mock_download, we start the timer at the beginning.
162 However, for the async version, we do the following:
163
164 First, we replace the return statement by a callback invocation. Normally
165 null would be an error message but we're not doing any error checking here
166 just yet.
167
168     return inst.delay_ms -> cb(null, inst.delay_ms)
169
170 Then we pull the tm.elapsed() call into the delay function. It is exactly
171 this call (along with the callback) which is delayed by inst.delay_ms in the
172 setTimeout.
173
174 The big difference from the synchronous_mock_download function is that
175 we need to explicitly engineer certain lines of code to complete before
176 other lines of code.
177 */
178 var asynchronous_mock_download = function(inst, cb) {
179     var tm = new Timer('' + inst.url);
180     tm.start();
181     var delayfn = function() {

```

```

182         tm.elapsed();
183         cb(null, inst.delay_ms);
184     };
185     setTimeout(delayfn, inst.delay_ms);
186 };
187
188 /**
189  Restructures the synchronous_example2 to be asynchronous. Note that the
190  whole trick is in structuring code such that you ensure that certain
191  lines occur after other lines - e.g. tm.elapsed() should not occur
192  before summarize(results) can occur.
193 */
194 var asynchronous_example = function(insts) {
195     var tm = new Timer('Asynchronous');
196     tm.start();
197
198     var async_summarize = function(err, results) {
199         summarize(results);
200         tm.elapsed();
201     };
202
203     async.map(insts, asynchronous_mock_download, async_summarize);
204 };
205
206
207 /**
208  Finally, the main routine itself is just a simple wrapper that
209  we use to group and isolate code.
210 */
211 var main = function() {
212     var nn = 5;
213     var insts = build_insts(nn);
214     synchronous_example(insts);
215     asynchronous_example(insts);
216 };
217
218 main();

```

You might think the async examples are more complicated. And it is true that for many simple scripts, the use of asynchronous coding can be considered a performance improvement that adds unnecessary complexity. You now need to think in terms of callbacks rather than return statements, you don't know when a given function will return for certain, and you generally need to give much more thought to the order of operations in your program than you did before. Maybe this just isn't worth it when you just want to download some URLs to disk.

However, it's much easier to turn an asynchronous program into a synchronous one than vice versa. Moreover, for any application that involves realtime updates (like chat, video

chat, video games, or monitoring) or sending data across the network (like a webapp!), you'll find it anywhere from helpful to critical to think asynchronously from the very beginning. Additionally, in other languages like Python, while it's possible to write asynchronous code with libraries like Twisted, all the core libraries are synchronous. This makes it more difficult to write new asynchronous code in other languages as you would need to launch said synchronous code in background processes (similar to the [bash ampersand](#)) to prevent your main routine from blocking. Finally, in Node the use of flow control libraries (like [async](#)), implementations of [Promises/A+](#) and [Functional Reactive Programming](#), and the forthcoming [yield](#) statement allow cleanup and organization of asynchronous code so that it's much easier to work with; see Amodeo's [excellent talk](#) for more perspective if you want to go further than [async](#).

All that said, it should be clear at this point why a server-side async language is the right tool at least some of the time. Ryan Dahl's [rationale](#) for doing Node specifically in JS was several-fold: first, that Javascript was very popular; second, that its asynchronous features were already heavily used in client side code (e.g. via [AJAX](#)); third, that despite this popularity and there was no popular server-side implementation of JS and hence no legacy synchronous library code; and fourth, that Google's then-new v8 JS interpreter provided an engine that could be extracted from Chrome and placed into a command line utility. Thus, Node was born, with the concept that Node libraries could be used in both the client and server and could be built from the ground up to be async by default. That's exactly what has happened over the last few years.

What are the advantages and disadvantages of Node?

Now that we understand the motivation for Node's asynchronous programming paradigm (namely, to minimize the impact of IO latency), we can start thinking through the [advantages](#) and disadvantages of Node:

- Advantages
 - Node.js is in Javascript, so you can [share code](#) between the frontend and backend of a web application. The potential for reducing code duplication, [increasing](#) the pool of available engineers, and improving¹ conceptual integrity is perhaps the single most important reason Node is worth learning. It is likely to become very popular in the medium-term, especially once new frameworks like Meteor (which are built on Node) start gaining in popularity. ([1](#), [2](#), [3](#)).
 - Node is inherently well-suited for:
 - * Working with existing protocols (HTTP, UDP, etc.), writing custom protocols or [customizing](#) existing ones. The combination of the Node standard library

¹For example, in the Django framework for Python, to implement something seemingly simple like an email validation, you would be checking that the syntax of the email satisfies a regular expression (often in JS on the client side for responsiveness) and that the email itself is not already present in a database (in Python code on the server side). Either of these would raise a different error message which you would return and process in JS to update the page, displaying some kind of [red error message](#). Try [signing up](#) for a Yahoo account to see an example. The issue here is that something which is conceptually simple (validating an email) can have its logic split between two different languages on the client and server. But with the new full-stack JS frameworks like Meteor, built on top of Node, you can simply do something like `Meteor.isClient` and `Meteor.isServer` ([example](#)) and put all the email validation logic in the same place, executing the appropriate parts in the appropriate context.

and the support for asynchrony/event-based programming make it natural to think of your app in terms of protocols and [networked components](#).

- * Soft realtime apps², like [chat](#) or [dashboards](#). Combining Node on the backend with something like [Angular.js](#) on the frontend can enable sophisticated event-based apps that update widgets very rapidly as new data comes in.
- * [JSON APIs](#) that essentially wrap a database and serve up JSON in response to HTTP Requests.
- * Streaming apps, like transloadit's [realtime encoding](#) of video.
- * Full stack JS interactive webapps like [Medium.com](#) or the kinds of things you can build with the [Meteor](#) framework.

- Disadvantages

- Node is not suitable for heavy mathematics ([1](#), [2](#), [3](#)); you'd want to call an external C/C++ math library via [addons](#) or something like [node-ffi](#) (for “foreign function interface”) if you need low-latency math within your main program. If you can tolerate high-latency³, then a python webservice exposing [numpy/scipy](#) could also work (example).
- Node is fine for kicking off I/O heavy tasks, but you wouldn't want to implement them in Node itself; you probably want to use C/C++ for that kind of thing to get [low-level](#) access, or perhaps Go⁴.
- The idiomatic way of programming in Node via asynchronous callbacks is actually quite different from how you'd write the equivalent code in Python or Ruby, and requires significant conceptual adjustment ([1](#), [2](#), [3](#)), though with various flow control libraries (especially [async](#)) and the important new [yield](#) statement, you can code in Node in a manner similar to how you'd do functional programming in Python or Ruby.
- It's maturing rapidly, but Node doesn't yet have the depth of libraries of Python or Ruby.

It's also useful to keep in mind that the `node` binary is more similar to the `python` or `ruby` binaries, in that it is a server-side interpreter⁵ with a set of standard libraries. The the web framework on top of Node is a distinct entity in its own right. Indeed, it is likely that the

²You wouldn't want to use Node for [hard realtime](#) apps, like the controller for a car engine, due to the fact that JS is [garbage collected](#) and thus can cause intermittent pauses. It is true that the v8 JS engine that Node is based on now uses an [incremental](#) garbage collector (GC) rather than a [stop the world](#) GC, but it is still fair to say that real-time apps in garbage-collected languages are a subject of [active research](#), and that you'd probably want to program that car engine controller in something like C with the PREEMPT_RT patch that turns Linux into a realtime OS ([1](#), [2](#)).

³By “tolerate high latency” we mean you can tolerate a delay between when you invoke the math routine and when you get results. For example, this would be acceptable for offline video processing, but would not be acceptable for doing realtime graphics.

⁴You should probably use C/C++, though, if you're already using Node. The reason is that you don't want to get too experimental. Node is vastly more stable than it was a year or two ago, and is used for [real sites](#) now, but you want to limit the number of experimental technologies in your stack.

⁵If we want to be precise: the `node` binary is not the same thing as the abstract Javascript programming language (specified by the [grammar](#) in the ECMAScript specification), in the same way that the [CPython](#) binary named `python` is not the same thing as the abstract Python programming language (which is specified by a [grammar](#)).

ultimate successor to Ruby/Rails or Python/Django will be based on Node.js. There are a few candidates for this successor:

- The so-called **MEAN** stack (MongoDB, Express, Angular, and Node), as a pseudo-successor⁶ to the LAMP stack (Linux, Apache, MySQL, and PHP).
- A realtime JS framework like [Meteor](#) or [Derby](#). Meteor in particular is backed by significant venture capital and is worth checking out (do try the [examples](#) here). When it attains full maturity, it will likely become quite popular.
- It's also worth mentioning the [Matador](#) framework from Medium.com by the founder of Blogger and Twitter; this is one of the first large sites [built on](#) Node (here's [more](#))

For this class we'll be using three out of four components of the MEAN stack: E ([Express](#)), A ([Angular](#)), and N ([Node](#)). Rather than [MongoDB](#) and the associated [Mongoose](#) module, however, we'll use the somewhat more conservative choice of [PostgreSQL](#) and the [sequelize](#) module for our database. Overall, Node is maturing rapidly and being used in [production](#) at an increasing number of companies, so it's worth experimenting with while keeping in mind that it has more in the way of rough edges than older platforms.

Asynchronous Programming and Flow Control

Now that we understand what Node is suited for, let's extend this and do a more in-depth example of asynchronous programming. We want to do something seemingly simple: hit the Node documentation site, get a list of all modules, and group these modules by their stability. Adapting an example from [Takada's](#) book, in pseudocode this would look something like this:

```
1 for(var i = 0; i < docurls; i++) {  
2     request(docurls[ii], function(err, resp, body) {  
3         // Parse the response and response body  
4     });  
5 }  
6 after_url_downloading();
```

This code is very simple when synchronous: loop over a bunch of URLs, download them, and then move on to the next thing. However, this seemingly simple `for` loop introduces at least three new things to keep in mind when the internal `request` calls become asynchronous:

1. First, we need to wait for all the `request` calls to complete before executing `after_url_downloading`.

⁶We say [pseudo-successor](#) because the individual terms in the acronym don't map one-to-one. Specifically, (1) Linux and/or BSD are assumed today as the OS, whereas they weren't in the early 2000s when the LAMP acronym was coined (as many were still using Windows). That is, the MEAN stack is assumed to be deployed on Linux and often developed on a Mac. (2) Apache would potentially be replaced by something like [nginx](#) rather than Node directly. (3) PHP in the past was both a server-side and client-side language, whereas Node is the basic JS interpreter, Express is the backend web framework, and Angular is the frontend web framework. (4) MongoDB could be considered a replacement for MySQL, and the MongoDB/mongoose combination is indeed well debugged and allows you to use JS within the database as well, but you also want to consider using PostgreSQL/sequelize, because PostgreSQL is more stable/mature than MongoDB. That said, MEAN is an interesting acronym and tech stack that looks likely to gain some traction.

2. Second, we might need to limit the number of parallel `request` calls if `docurls` is a very long list (e.g. 1000s of URLs). You can do `ulimit -n` to determine the number of simultaneous filehandles that your OS supports; see [here](#) for more details.
3. Third, we will often want to pass the results of these downloads to the next function, and right now the callback in each `request` isn't populating a data structure.

To solve these issues we need:

1. A way to force all the `requests` to complete before `after_url_downloading`
2. A way to determine when all `request` invocations been completed (e.g. by setting a flag)
3. A way to accumulate the results of the requests into a datastructure for use by subsequent code
4. A way to limit the parallelization of the number of requests

Let's take a look at the following code, which solves these issues in order to download the Node documentation and output the list of modules grouped by stability. First, [download](#) and execute the script on an EC2 instance by doing this:

```

1 wget https://d396qusza40orc.cloudfront.net/startup/code/list-stable-modules.js
2 npm install underscore async request
3 node list-stable-modules.js

```

When you run the script it should look like Figure 2.

```
[ubuntu@ip-172-31-28-87:~]$node list-stable-modules.js
{
  "1": [
    "documentation",
    "cluster"
  ],
  "2": [
    "crypto",
    "domain",
    "punycode",
    "readline",
    "stream",
    "tty",
    "vm"
  ],
  "3": [
    "buffer",
    "child_process",
    "debugger",
    "dns",
    "fs",
    "http",
    "https",
    "net",
    "path",
    "querystring",
    "string_decoder",
    "tls",
    "dgram",
    "url",
    "zlib"
  ],
  "4": [
    "console",
    "events",
    "os"
  ],
  "5": [
    "assert",
    "modules",
    "timers",
    "util"
  ],
  "undefined": [
    "synopsis",
    "addons",
    "globals",
    "process",
    "repl"
  ]
}
```

Figure 2: The script hits the nodejs.org website, downloads all the documentation in JSON, and outputs modules by their degree of stability.

Now let's see how the code works. At a high level it does the following:

1. Downloads <http://nodejs.org/api/index.json>
2. Extracts all modules and creates a list of module URLs (e.g. <http://nodejs.org/api/fs.json>)
3. Downloads each module URL JSON asynchronously, launching dozens of simultaneous requests
4. Parses the JSON, determines the **stability index** of each module, and organizes modules by stability
5. Prints this **stability_to_names** data structure to the console

Read through the following commented source code for more details. Our implementation relies crucially on the use of the `async` library, perhaps the [most popular](#) way in Node to manage asynchronous flow control. The basic concept is to develop functions that accomplish the above five tasks in isolation. As we get each function to work, we save its output to a variable with the simple `save` debugging utility.

```
1  /*
2   * 0. Motivation.
3   *
4   Try executing this script at the command line:
5
6     node list-stable-modules.js
7
8   The node.js documentation is partially machine-readable, and has
9   "stability indexes" as described here, measuring the stability and
10  reliability of each module on a 0 (deprecated) to 5 (locked) scale:
11
12  http://nodejs.org/api/documentation.html#documentation_stability_index
13
14  The main index of the node documentation is here:
15
16  http://nodejs.org/api/index.json
17
18  And here is a sample URL for the JSON version of a particular module's docs:
19
20  http://nodejs.org/api/fs.json
21
22  Our goal is to get the node documentation index JSON, list all modules,
23  download the JSON for those modules in parallel, and finally sort and
24  output the modules by their respective stabilities. To do this we'll use
25  the async flow control library (github.com/caolan/async).
26
27  As a preliminary, install the following packages at the command line:
28
29  npm install underscore async request;
```

```

30
31 Now let's get started.
32
33 */
34 var uu = require('underscore');
35 var async = require('async');
36 var request = require('request');
37
38 /*
39 1. Debugging utilities: save and log.
40
41 When debugging code involving callbacks, it's useful to be able to save
42 intermediate variables and inspect them in the REPL. Using the code below,
43 if you set DEBUG = true, then at any point within a function you can put
44 an invocation like this to save a variable to the global namespace:
45
46     save(mod_data, 'foo')
47
48 For example:
49
50     function mod_urls2mod_datas(mod_urls, cb) {
51         log(argumentscallee.name);
52         save(mod_urls, 'foo'); // ----- Note the save function
53         async.map(mod_urls, mod_url2mod_data, cb);
54     }
55
56 Then execute the final composed function, e.g. index_url2stability_to_names, or
57 otherwise get mod_urls2mod_datas to execute. You can confirm that it
58 has executed with the log(argumentscallee.name) command.
59
60 Now, at the REPL, you can type this:
61
62     > mod_urls = global._data.foo
63
64 This gives you a data structure that you can explore in the REPL. This is
65 invaluable when building up functions that are parsing through dirty data.
66
67 Note that the DEBUG flag allows us to keep these log and save routines
68 within the body of a function, while still disabling them globally by
69 simply setting DEBUG = false. There are other ways to do this as well, but
70 this particular example is already fairly complex and we wanted to keep
71 the debugging part simple.
72 */
73 var _data = {};
74 var DEBUG = false;
75 var log = function(xx) {
76     if(DEBUG) {

```

```

77     console.log("%s at %s", xx, new Date());
78 }
79 };
80 function save(inst, name) {
81     if(DEBUG) { global._data[name] = inst; }
82 }
83
84 /*
85 NOTE: Skip to part 6 at the bottom and then read upwards. For
86 organizational reasons we need to define pipeline functions from the last
87 to the first, so that we can reference them all at the end in the
88 async.compose invocation
89
90 2. Parse module data to pull out stabilities.
91
92 The mod_data2modname_stability function is very messy because the
93 nodejs.org website has several different ways to record the stability of a
94 module in the accompanying JSON.
95
96 As for mod_datas2stability_to_names, that takes in the parsed data
97 structure built from each JSON URL (like http://nodejs.org/api/fs.json)
98 and extracts the (module_name, stability) pairs into an object. The for
99 loop groups names by stability into stability_to_names. Note that we use
100 the convention of a_to_b for a dictionary that maps items of class a to
101 items of class b, and we use x2y for a function that computes items of
102 type y from items of type x.
103
104 The final stability_to_names data structure is the goal of this script
105 and is passed to the final callback stability_to_names2console at
106 the end of async.compose (see the very end of this file).
107 */
108 function mod_data2modname_stability(mod_data) {
109     var crypto_regex = /crypto/;
110     var stability_regex = /Stability: (\d)/;
111     var name_regex = /doc\api\(\w+\).markdown/;
112     var modname = name_regex.exec(mod_data.source)[1];
113     var stability;
114     try {
115         if(crypto_regex.test(modname)) {
116             var stmp = stability_regex.exec(mod_data.modules[0].desc)[1];
117             stability = parseInt(stmp, 10);
118         }
119         else if(uu.has(mod_data, 'stability')) {
120             stability = mod_data.stability;
121         }
122         else if(uu.has(mod_data, 'mics')) {
123             stability = mod_data.mics[0].mics[1].stability;

```

```

124     }
125     else if(uu.has(mod_data, 'modules')) {
126         stability = mod_data.modules[0].stability;
127     }
128     else if(uu.has(mod_data, 'globals')) {
129         stability = mod_data.globals[0].stability;
130     } else {
131         stability = undefined;
132     }
133 }
134 catch(e) {
135     stability = undefined;
136 }
137 return {"modname": modname, "stability": stability};
138 }

139
140 function mod_datas2stability_to_names(mod_datas, cb) {
141     log(mod_datas);
142     log(arguments.callee.name);
143     modname_stabilities = uu.map(mod_datas, mod_data2modname_stability);
144     var stability_to_names = {};
145     for(var ii in modname_stabilities) {
146         var ms = modname_stabilities[ii];
147         var nm = ms.modname;
148         if(uu.has(stability_to_names, ms.stability)) {
149             stability_to_names[ms.stability].push(nm);
150         } else{
151             stability_to_names[ms.stability] = [nm];
152         }
153     }
154     cb(null, stability_to_names);
155 }
156
157 /*
158 3. Download module urls and convert JSON into internal data.
159
160 Here, we have a function mod_url2mod_data which is identical to
161 index_url2index_data, down to the JSON parsing. We keep it distinct for
162 didactic purposes but we could easily make these into the same function.
163
164 Note also the use of async.mapLimit to apply this function in parallel to
165 all the mod_urls, and feed the result to the callback (which we can
166 leave unspecified in this case due to how async.compose works).
167
168 We use mapLimit for didactic purposes here, as 36 simultaneous downloads
169 is well within the capacity of most operating systems. You can use ulimit
170 -n to get one constraint (the number of simultaneous open filehandles in

```

```

171  Ubuntu) and could parse that to dynamically set this limit on a given
172  machine. You could also modify this to take a command line parameter.
173 /*
174 function mod_url2mod_data(mod_url, cb) {
175   log(arguments.callee.name);
176   var err_resp_body2mod_data = function(err, resp, body) {
177     if(!err && resp.statusCode == 200) {
178       var mod_data = JSON.parse(body);
179       cb(null, mod_data);
180     }
181   };
182   request(mod_url, err_resp_body2mod_data);
183 }
184
185 function mod_urls2mod_datas(mod_urls, cb) {
186   log(arguments.callee.name);
187   var NUM_SIMULTANEOUS_DOWNLOADS = 36; // Purely for illustration
188   async.mapLimit(mod_urls, NUM_SIMULTANEOUS_DOWNLOADS, mod_url2mod_data, cb);
189 }
190
191 /*
192 4. Build module URLs (e.g. http://nodejs.org/api/fs.json) from the JSON
193 data structure formed from http://nodejs.org/api/index.json.
194
195 The internal modname2mod_url could be factored outside of this function,
196 but we keep it internal for conceptual simplicity.
197 */
198 function index_data2mod_urls(index_data, cb) {
199   log(arguments.callee.name);
200   var notUndefined = function(xx) { return !uu.isUndefined(xx); };
201   var modnames = uu.filter(uu.pluck(index_data.desc, 'text'), notUndefined);
202   var modname2mod_url = function(modname) {
203     var modregex = /\[(\[(^\])+\]\]\(([\^\])+\).html\)/;
204     var shortname = modregex.exec(modname)[2];
205     return 'http://nodejs.org/api/' + shortname + '.json';
206   };
207   var mod_urls = uu.map(modnames, modname2mod_url);
208   cb(null, mod_urls);
209 }
210
211 /*
212 5. Given the index_url (http://nodejs.org/api/index.json), pull
213 down the body and parse the JSON into a data structure (index_data).
214
215 Note that we could factor out the internal function with some effort,
216 but it's more clear to just have it take the callback as a closure.
217

```

```

218 Note also that we define the function in the standard way rather than
219 assigning it to a variable, so that we can do log(arguments.callee.name).
220 This is useful to follow which async functions are being executed and
221 when.
222
223 */
224 function index_url2index_data(index_url, cb) {
225   log(arguments.callee.name);
226   var err_resp_body2index_data = function(err, resp, body) {
227     if(!err && resp.statusCode == 200) {
228       var index_data = JSON.parse(body);
229       cb(null, index_data);
230     }
231   };
232   request(index_url, err_resp_body2index_data);
233 }
234
235 /*
236 6. The primary workhorse async.compose (github.com/caolan/async#compose)
237 sets up the entire pipeline in terms of five functions:
238
239 - stability_to_names2console // Print modules ordered by stability to console
240 - mod_datas2stability_to_names // List of modules -> ordered by stability
241 - mod_urls2mod_datas // nodejs.org/api/MODULE.json -> List of module data
242 - index_data2mod_urls // Extract URLs of module documentation from index_data
243 - index_url2index_data // Get nodejs.org/api/index.json -> JSON index_data
244
245 The naming convention here is a useful one, especially at the beginning of
246 a program when working out the data flow. Let's understand this in terms
247 of the synchronous version, and then the async version.
248
249 Understanding the synchronous version
250 -----
251 If this was a synchronous program, we would conceptually call these
252 functions in order like so:
253
254   index_data = index_url2index_data(index_url)
255   mod_urls = index_data2mod_urls(index_data)
256   mod_datas = mod_urls2mod_datas(mod_urls)
257   stability_to_names = mod_datas2stability_to_names(mod_datas)
258   stability_to_names2console(stability_to_names)
259
260 Or, if we did it all in one nested function call:
261
262   stability_to_names2console(
263     mod_datas2stability_to_names(
264       mod_urls2mod_datas(

```

```
265     index_data2mod_urls(  
266         index_url2index_data(index_url))))  
267  
268 That's a little verbose, so we could instead write it like this:  
269
```

```
270     index_url2console = compose(stability_to_names2console,  
271                                 mod_datas2stability_to_names,  
272                                 mod_urls2mod_datas,  
273                                 index_data2mod_urls,  
274                                 index_url2index_data)  
275     index_url2console(index_url)  
276  
277 This is a very elegant and powerful way to represent complex dataflows,  
278 from web crawlers to genome sequencing pipelines. In particular, this  
279 final composed function can be exposed via the exports command.  
280
```

281 Understanding the asynchronous version

283 The main difference between the synchronous and asynchronous versions is
284 that the synchronous functions would each **return** a value on their last
285 line, while the asynchronous functions do not directly return a value but
286 instead pass the value directly to a callback. Specifically, every time
287 you see this:

```
289     function foo2bar(foo) {  
290         // Compute bar from foo, handling errors locally  
291         return bar;  
292     }
```

294 You would replace it with this, where we pass in a callback bar2baz:

```
296     function foo2bar(foo, bar2baz) {  
297         // Compute bar from foo  
298         // Pass bar (and any error) off to the next function  
299         bar2baz(err, bar);  
300     }
```

302 So to compose our five functions with callbacks, rather than do a
303 synchronous compose, we use the `async.compose` function from the `async`
304 library:

```
306     https://github.com/caolan/async#compose
```

308 The concept is that, like in the synchronous case, we effectively
309 generate a single function `index_url2console` which represents
310 the entire logic of the program, and then feed that `index_url`.

```

312 */
313 function stability_to_names2console(err, stability_to_names) {
314   log(arguments.callee.name);
315   console.log(JSON.stringify(stability_to_names, null, 2));
316 }
317
318 var index_url2console = async.compose(mod_datas2stability_to_names,
319                                         mod_urls2mod_datas,
320                                         index_data2mod_urls,
321                                         index_url2index_data);
322
323 var index_url = "http://nodejs.org/api/index.json";
324 index_url2console(index_url, stability_to_names2console);

```

Note that the code solves all four of the problems that we mentioned:

1. A way to force all the `requests` to complete before subsequent code was executed
 - We did this by using `async.compose` to force a given set of downloads to execute before another segment began (e.g. the function `mod_urls2mod_datas` ran before `mod_datas2stability_to_names` was invoked)
2. A way to determine when all `request` invocations been completed (e.g. by setting a flag)
 - Again, we did this with the `async.compose` structure. If you really wanted just a boolean flag, you could have combined `async.map` and `async.every`.
3. A way to accumulate the results of the requests into a datastructure for use by subsequent code
 - The `async.compose` function helped us out again, as the output of one stage was fed directly into the next. Alternatives here would be `async.waterfall`, `async.series`, or `async.map`.
4. A way to limit the parallelization of the number of requests
 - Here we used `async.mapLimit` to put an upper bound on the number of parallel downloads.

In general, `async.compose` is one of the cleanest ways to write server-side asynchronous JS code, especially for anything that can be reduced to a dataflow pipeline. One of the benefits of the compose approach is that you can write individual functions and debug them by themselves; another is that once you have the final composed function working, you can use `module.exports` to expose that alone to another module for import (e.g. in this case `index_url2console` would be the sole exported function).

As a useful exercise to test whether you understand this code, you might write a simple command line script that hits this [Federal Register API JSON URL](#), builds up data structures for each listed agency, organizes them in a tree hierarchy as specified by the `parent_id` field, and then outputs the result to the command line.

Basic debugging with the Node Debugger

Now that we understand latency and have looked at a medium-size asynchronous script in detail, it's worth learning a bit more about debugging. To know how to debug code with confidence in a language is to understand that language. Let's do a quick example with the Node debugger. Suppose that we boot up an HTTP server as shown:

```
1 wget https://d396qusza40orc.cloudfront.net/startup/code/debugger-examples.js
2 node debugger-examples.js      # start the HTTP server
3 node debug debugger-examples.js # to invoke for debugging; see text
```

The source code is shown below:

```
1 #!/usr/bin/env node
2
3 var http = require('http');
4 var counter = 0;
5 var serv = http.createServer(function(req, res) {
6     counter += 1;
7     res.writeHead(200);
8     debugger;
9     res.end("Cumulative number of requests: " + counter);
10 });
11 var port = 8080;
12 serv.listen(port);
13 console.log("Listening at %s", port);
```

The goal of this server is simply to display the cumulative number of HTTP requests that have been made to the server over time. Our expectation is that as we refresh the page from any computer, that this number will increase. But if you [download](#) the script, run it with `node debugger-examples.js`, and view the results in Chrome, you will see something surprising. The counter increments by two rather than by one with each refresh. We can debug this with the built-in Node debugger, by doing `node debug debugger-examples.js`. See Figures 3-11 for a worked example.

Debugging Node code

This simple example illustrates how to debug the output produced by a Node HTTP server running on an EC2 instance.

- ① Download code (see text), determine the EC2 hostname, and run the script with: `node debugger-examples.js`

```
[ubuntu@ip-172-31-28-87:~]$wget https://d396qusza40orc.cloudfront.net/startup%2Fcode%2Fdebugger-examples.js -O debugger-examples.js
--2013-08-20 21:21:50-- https://d396qusza40orc.cloudfront.net/startup%2Fcode%2Fdebugger-examples.js
Resolving d396qusza40orc.cloudfront.net (d396qusza40orc.cloudfront.net)... 54.230.68.12, 54.230.68.147, 54.230.68.188, ...
Connecting to d396qusza40orc.cloudfront.net (d396qusza40orc.cloudfront.net)|54.230.68.121:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 318 [text/javascript]
Saving to: 'debugger-examples.js'

100%[=====] 318
2013-08-20 21:21:50 (69.3 MB/s) - `debugger-examples.js' saved [318/318]

[ubuntu@ip-172-31-28-87:~]$curl http://169.254.169.254/latest/meta-data/public-hostname; echo ""
ec2-54-213-98-94.us-west-2.compute.amazonaws.com
[ubuntu@ip-172-31-28-87:~]$node debugger-examples.js
Listening at 127.0.0.1:8080
```

Figure 3: First, download `debugger-examples.js` and determine the current EC2 hostname. Then invoke `node debugger-examples.js` at the command line to start the HTTP server.

- ② View in Chrome and refresh a few times. Note mysterious bug: why is the count incrementing by two?

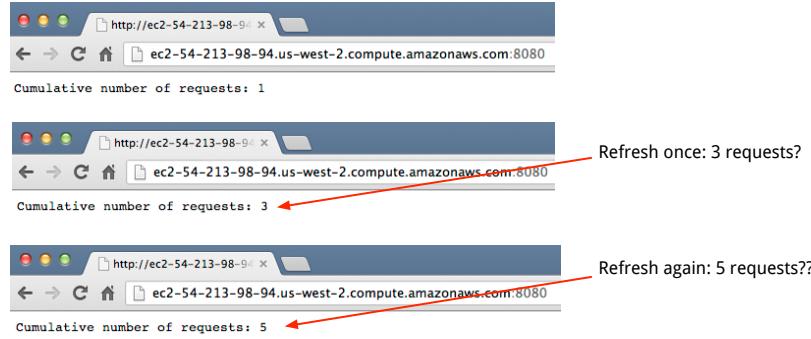


Figure 4: Now open up Chrome and refresh a few times. You will see that the number of requests increments by two rather than one, which is surprising.

- ③ Disable the rlwrap wrapper by using the bash alias command (as shown) or by commenting out the corresponding line in your `~/.bashrc` file. This wrapper can be convenient, but will interfere with the ability to quit the repl in the debugger without exiting the whole thing.

```
[ubuntu@ip-172-31-28-87:~]$alias l grep node
alias node='env NODE_NO_READLINE=1 rlwrap node'
alias node_repl='node -e "require('repl').start({ignoreUndefined: true})"'
[ubuntu@ip-172-31-28-87:~]$alias node=node'
[ubuntu@ip-172-31-28-87:~]$alias l grep node
alias node='node'
alias node_repl='node -e "require('repl').start({ignoreUndefined: true})"'
```

Figure 5: To use the debugger we'll need to disable the `rlwrap` wrapper around `node`. This is recommended by the Node documentation as it provides a better REPL for most purposes, but will interfere with the debugger's internal REPL.

- ④ Then restart the code in the debugger with `node debug debugger-examples.js`. Type `cont` to make it run until the first breakpoint is hit, which is where "debugger;" was present in the code. The breakpoint will be triggered when an HTTP request is submitted by Chrome.

```
[ubuntu@ip-172-31-28-87:~]$node debug debugger-examples.js
< debugger listening on port 5858
connecting... ok
break in debugger-examples.js:3
1
2
3 var http = require('http');
4 var counter = 0;
5 var serv = http.createServer(function(req, res) {
debug> cont ←
< Listening at 127.0.0.1:8080
debug> █
```

After we start the debugger, type `cont` to run the code and wait for a request to be received by the HTTP server

Figure 6: Now we quit `node debugger-examples.js` and restart it as `node debug debugger-examples.js`, as shown. We then type `cont` to continue running the script until a breakpoint is reached, at which point we return control to the debugger.

- ⑤ Navigate to the same URL you did before. Chrome will hang, showing that the HTTP request has been submitted but an HTTP response has not yet been generated.



Browser hangs on first request
as expected; now enter debugger

Figure 7: Now we go to Chrome and artificially induce a breakpoint by connecting to the remote server. The browser will hang as an HTTP request has been set but no HTTP response has yet been received (because the code is halted at the breakpoint in the debugger, waiting for your input).

- ⑥ Return to the command line. The debugger has placed you at the breakpoint. Look at the local variable values with the `repl` command and hit Ctrl-C once to return to the debugger. (If you didn't do the alias step above, this Ctrl-C may exit the whole thing.) The first time through, everything looks ok. Let's do `cont` again to resume.

```
[ubuntu@ip-172-31-28-87:~]$node debug debugger-examples.js
< debugger listening on port 5858
connecting... ok
break in debugger-examples.js:3
  1
  2
  3 var http = require('http');
  4 var counter = 0;
  5 var serv = http.createServer(function(req, res) {
debug> cont
< Listening at 127.0.0.1:8080
break in debugger-examples.js:8
  6   counter += 1;
  7   res.writeHead(200);
  8   debugger;
  9   res.end("Cumulative number of requests: " + counter);
10 });
debug> repl
Press Ctrl + C to leave debug repl
> counter
1
> Object.keys(req)
[ 'socket',
  'connection',
  'httpVersion',
  'complete',
  'headers',
  'trailers',
  'readable',
  'url',
  'method',
  'statusCode',
  'client',
  'httpVersionMajor',
  'httpVersionMinor',
  'upgrade' ]
> req.headers
{ 'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.57 Safari/537.36',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'accept-encoding': 'gzip,deflate,sdch',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-language': 'en-US,en;q=0.8',
  host: 'ec2-54-213-98-94.us-west-2.compute.amazonaws.com:8080' }
> req.url
 '/'
debug> cont
```

The debugger has now hit a breakpoint (line 8 where it says "debugger;" in the source). We type "repl" to inspect the local variables.

Here we're just printing the value of the counter, which has been incremented to 1.

While in the repl, we can print the keys of the "req" object, containing the HTTP request, and look at the req.headers field and req.url field in particular. In this manner we can look at the values of variables within live, running code.

Everything looks good on the first request. Now we're hitting cont. This returns the HTTP response and the debugger will wait for input.

Figure 8: We return to the debugger and run the `repl` command at the breakpoint. We poke around and print a few variables as shown. Everything looks in order so we exit the internal `repl` with Ctrl-C and resume execution with `cont`. This releases the HTTP response.

- ⑦ Now that the HTTP response has been generated, Chrome is no longer hanging. Refresh the page and Chrome will hang once more.



Figure 9: Now when we return to Chrome, the HTTP response has been received and displayed. Let's try refreshing again to see if we can reproduce the oddity where the counter is incrementing by two each time.

⑧ Return to the debugger and enter the `repl` command once more. Now print out the counter and the fields of the request. Aha! We see that Chrome has made a request for `/favicon.ico`. That is, it's not just requesting the base URL, but the "favicon" used for representing a site in a Favorite/Bookmark and in each tab. So we are effectively double counting each connection from Chrome.

```

> req.url ← Here's the print statement from the previous
'/' bit of the debugger
debug> cont ← After the refresh, the breakpoint
break in debugger-examples.js:8 on line 8 is hit again and the debugger
       returns control
  6   counter += 1;
  7   res.writeHead(200);
  8   debugger;
  9   res.end("Cumulative number of requests: " + counter);
10 };
debug> repl ← Now we enter the repl
Press Ctrl + C to leave debug repl
> counter
2
> req.headers
{ 'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/537.36 (KHTML, like Gecko... (length: 119)', ← When inspecting the req.url we see
connection: 'keep-alive', the issue. Chrome is making a request
'accept-encoding': 'gzip,deflate,sdch', for "/favicon.ico", not just "/".
accept: '*/*',
'accept-language': 'en-US,en;q=0.8',
host: 'ec2-54-213-98-94.us-west-2.compute.amazonaws.com:8080' }
> req.url ←
'/favicon.ico' ←

```

Figure 10: Within the debugger again, we again do the `repl` command and look at the counter. It's only incrementing by 1, to the value 2. So what's going on? Aha, if we look at the `req` data structure in more detail, the `req.url` field is `/favicon.ico` rather than `/`. So two HTTP requests are being initiated rather than one by Chrome. And that is happening very fast and imperceptibly, so it seems like the counter is increasing by two each time.

⑨ One way to fix this bug is by including a simple "if" statement in the code that only increments for those requests that are not for the favicon file. If you were interested in tracking page loads, you'd need to do something more sophisticated, as a single page load could result in many HTTP requests for CSS, JS, and image files (not just favicons).

```

#!/usr/bin/env node

var http = require('http');
var counter = 0;
var serv = http.createServer(function(req, res) {
  if(req.url != "/favicon.ico") {
    counter += 1;
    res.writeHead(200);
    res.end("Cumulative number of requests: " + counter);
  }
});
var port = 8080;
serv.listen(port);
console.log("Listening at 127.0.0.1:%s", port); ← Here's a simple conditional test
                                         that filters out favicon.ico requests
                                         when counting the number of
                                         page loads.

```

Figure 11: Now that we understand what is causing the bug, we can address it by including a simple `if` statement in our code.

This simple example illustrates several things:

- How to create breakpoints by including `debugger` statements
- How to debug a network application by initiating a request and jumping into the function that is processing that request to generate a response on the other side
- The kinds of issues that arise in web applications (often they can be reduced to an “unexpected response for this request”)

Now, you can get much more sophisticated in terms of debugging.

1. First, you don't actually need to include `debugger` statements in the source code, as you can [create](#) and delete breakpoints in the debugger via `setBreakpoint/sb` and `clearBreakpoint/cb`.
2. Second, you can combine the use of the `debugger` with logging statements, e.g. via [console.log](#).
3. Third, you can use something like the [node-inspector](#) module for accomplishing many of the same tasks in a web interface (you may find the command line debugger we just reviewed to be faster for most purposes).
4. Fourth, you can use [Eclipse](#) or [WebStorm](#) if you want a fancier UI than the command line debugger.
5. Finally, and perhaps most importantly, you can define a set of custom debugging functions like the `save` and `log` functions as we did in the [list-stable-modules.js](#) script. Any reasonably sophisticated application should have quite a few of these custom debugging tools, which you want to co-develop along with your data structures and functions.

Overall, the `debugger` command and `console.log` along with some custom functions should be sufficient for most backend debugging. The advantage of command line debugging combined with logging is its extreme speed and portability, but do feel free to experiment with these other options.

Summary and Recap

To summarize, in this lecture we went through the concept of latency, looked at sample code to understand the importance of async vs. sync coding, looked at a medium-size script to get a sense of how a command line script involving asynchrony could be developed, and finally got to learn about several techniques to debug Node code.