The No Bullshit Bible : Creating Web 2.0 Startups & Programming

Written by James Gillmore
Edited by Holly Welch

FaceySpacey

www.faceyspacey.com

Speccing

TABLE OF CONTENTS

Non-Technical

Chapter 7

7-1 SPECCING PRODUCT SPECCING | OVERVIEW OF HOW WE SPEC AT FACEYSPACEY

There are several schools of thought when it comes to how much you pre-plan your application and how much your developers develop it with agility, i.e. according to so-called “agile” practices. Our take is that “Agile” has been misinterpreted by many a failed startup as an excuse to operate without a proper plan.

However it goes deeper than that: we believe there are different best practices depending on the scenario. If you’re a funded startup (i.e. with funding in the millions), you have the luxury of refining an iterative process where you explore what you want to develop in small steps. You also have more experienced developers that often have worked together who have already mastered their iterative process. For new startups, this is a costly phase where you figure out your development process. So if you only have $50k-150k for your project, you’re in a completely different boat, and you simply cannot afford that. You have to operate completely differently. We believe it boils down to 1 thing you absolutely must do differently: completely plan what you will get for your $50-150k. This means you means you must know exactly what you plan to launch to the public in your initial offering. And this requires a greater degree of what has been often, and pejoratively, called “crystal ball” planning. This is where we excel at FaceySpacey and what we’d like to teach you. The process can of course be applied if you’re fortunate enough to be able to iterate more. It’s just being able to imagine your product properly and at a granular level.

The speccing process we'll cover in the following tutorials includes these components:

1) SCOPE OPTIMIZATION

2) INTRO TO BUILDING PAGE LAYOUTS

3) FINDING INFLUENCES

4) ORGANIZED WRITTEN SPECS

5) TASK ORGANIZATION & BUGTRACKING (Fogbugz)

6) DATABASE DESIGN

7) APPLICATION ARCHITECTURE

8) SPRINT PLANNING (And More on Fogbugz)

7-2 SPECCING PRODUCT SPECCING | SCOPE OPTIMIZATION

Typically at Faceyspacey we keep the scope of a brand new startup small, but quite a bit larger than what you'd get with one iteration. One typical iteration wouldn't be what you feel is suitable to launch to the public. So we'd spec the startup down to what a multi-million dollar startup might accomplish in about 10 small iterations/sprints as they figure out their product. In short, we determine the concise set of features that capture the essence of how you want to present to the public, and make sure not to waste any energy and resources going farther than absolutely necessary.

So what that means you need to imagine a complete product from the beginning, but as small as absolutely possible. In the last phase of the Speccing process--the Sprint Planning phase--we'll break up the plan into 5-10 sprints.

Ok so how do you optimize scope realistically? First you pinpoint your single value proposition you must get right. Without this working perfectly with a great user experience, you know you have nothing. So for one of our past startups, SnackSquare, that value proposition was the delivery of SMS text messages to people nearby your local store front. For this startup, I won't lie, we actually didn't do what I'm about to say here, which is how we've come to learn how important this technique of scope optimization is. So I'm going to work my way backward pinpointing our mistakes, and then summarize what we would have done differently.

First off, we imagined absolutely everything possible the startup would need. This included a way for merchants to add multiple stores, ways to add multiple coupons that could each exist at multiple stores (i.e. a many-to-many relationship), tools to track customers across multiple stores they may visit (or be near) and the different coupons they may use at different stores, and ways (i.e. entities) to represent the proximity-based connections between customers and a store when they are near it, entities to represent campaigns across all the previous entities and ultimately a lot more.

So hopefully you get the idea that we had to model a lot of real-world entities and the connections between them. We would have been a lot better off if we dropped the concept of coupons all together and even campaigns, as that was what everyone else was doing, and not central to our core value proposition of SMS delivery.

Instead, we would have simply allowed you to add stores. You wouldn't create campaigns because it would be a given that a single global campaign was operating against all your stores. In terms of metrics, instead of tracking the performance of coupons, campaigns, stores and customers, we'd track just the aggregate performance of all your stores. We wouldn't even let you drill down to the metrics on a store by store basis or the metrics for each customer (i.e. how many times they frequent your stores). We'd focus on aggregate system-wide stats, and make sure those text messages go out immediately when a potential customer is near your store. And we'd even let you only set one message at a time that would go out, i.e. a global campaign. We wouldn't have done that whole thing where you can set the timeing and dates of multiple simultaneously-running campaigns. Maybe we'd let you set the dates and times, but that's where it would end, and it would be one message that goes out.

The reason reducing the scope so much was so important was because the technology to track customers was so damn complicated that we needed to put all our attention on that. We were completely at the mercy of changing/evolving APIs that were at the cutting edge of "geo" and just experimenting themselves with what data they provided. We had to track checkins on Foursquare, Facebook, and Twitter, all with various techniques tailored to each social network and how it produces geo-tagged checkins/tweets/updates with lat/lng coordinates, and as we went the APIs changed--usually for the better.

So anyway, that's an example that should highlight how to reduce the scope for your startup. To put it in more abstract terms that summarizes what we did, I'll explain it like this:

1) FEW CODE ENTITIES - *keep the real life entities you have to model in code down to a bare minimum*

2) PINPOINT SINGLE VALUE PROPOSITION *- find your core value proposition and understand technically all that's required to execute it at a very deep level before you start. It's really hard to go that deep into the tech specs of a project at the beginning, but if you can narrow it down to that one thing, it makes it a lot easier to go the mile in breaking down the technical ramifications.*

3) ONE MAJOR INTERFACE *- imagine only one key interface where all the magic happens. Don't have 2+ sick interfaces that do lots of cool stuff. So that means you can have a bunch of standard web pages for your account info, and even a graph/analytics page. But only have one page where your end user gets his value from the application. Everything else could basically not exist.*

This sort of thinking is completely in vogue right now in the "Techcrunch scene" as I like to call it, i.e. KISS ("keep it simple stupid"). And for very good reasons: it's really hard to imagine everything you need to do to get your application to launch. You'll save yourself a lot of heart-ache by putting your mind to doing just the minimum you need, i.e. get out that MVP ("minimal viable product"). Ingrain these acronyms in your head. It's the only way you'll succeed.

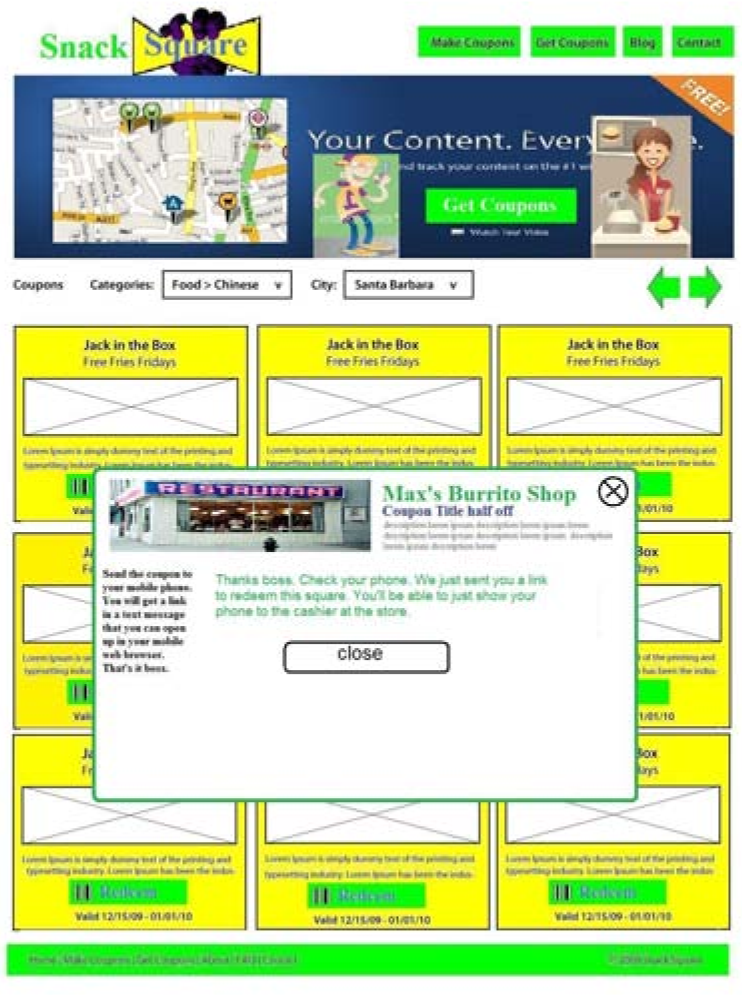
7-3 SPECCING PRODUCT SPECCING | INTRO TO BUILDING PAGE LAYOUTS

At FaceySpacey we produce layouts of every page of your application, and every state each page can be in. We do so in combination with a simultaneously running branding phase where each phase/aspect works off each other--we'll talk about the importance of graphic design while speccing later. The imagining of these layouts is what we do best and the most important first step in planning your applications. It's where you explore deep into what you're app could and should be.

Lots of startups have to build lots of potentially useless stuff in order to see far into what your application needs to be at launch. This can lead to wasting lots of money on developing code you ultimately won't use. So in essence you need to master being able to predict what you're application needs to become if you plan to save time and money.

To master this skill, you need to think about it like a process rather than an end result. It's not one set of layouts you make. Rather, it's many sets and revisions that evolve. Like, don't expect to do one round, or 3. Expect to be constantly evolving your layouts and learning more about your product through layouts. It's way cheaper to learn about your product in Photoshop or Microsoft Paint than it is in code. Explore all the possibilities for how you're app may look and function. Connect one interface to another. If you have a list of stores, create a page that lists the stores, put an edit/create button in the top right, then make the edit/create page, etc. Then on the edit/create page when you realize you must assign coupons to your stores, go make a coupon list page, and then a coupon/edit/create page, etc.

I'd like to say that if you've narrowed your scope well, the pages you need to create will be a small list, but the reality is that you're both narrowing/optimizing your scope (as described in the last tutorial) while making these layouts. You're constantly learning about what you're product is. If this phase goes successfully, you'll most likely have scrapped entire visions for your application, and removed many sub-concepts of each vision, etc. You may start out with one idea and totally change it to something more viable. During this phase you'll discover your core value proposition, and when you do you'll realize everything else that you have to scrap.



Send the coupon to your mobile phone. You will get a link in a text message that you can open up in your mobile web browser. That's it boss.

Thanks boss. Check your phone. We just sent you a link to redeem this square. You'll be able to just show your phone to the cashier at the store.

The goal is to get your application to a point where you can navigate through your screenshots and pretend like you're using your application. No button can't have a page missing for it, even if it leads to a modal popup. If it does, make another layout with the same page in the background, but the modal popup on top of it.

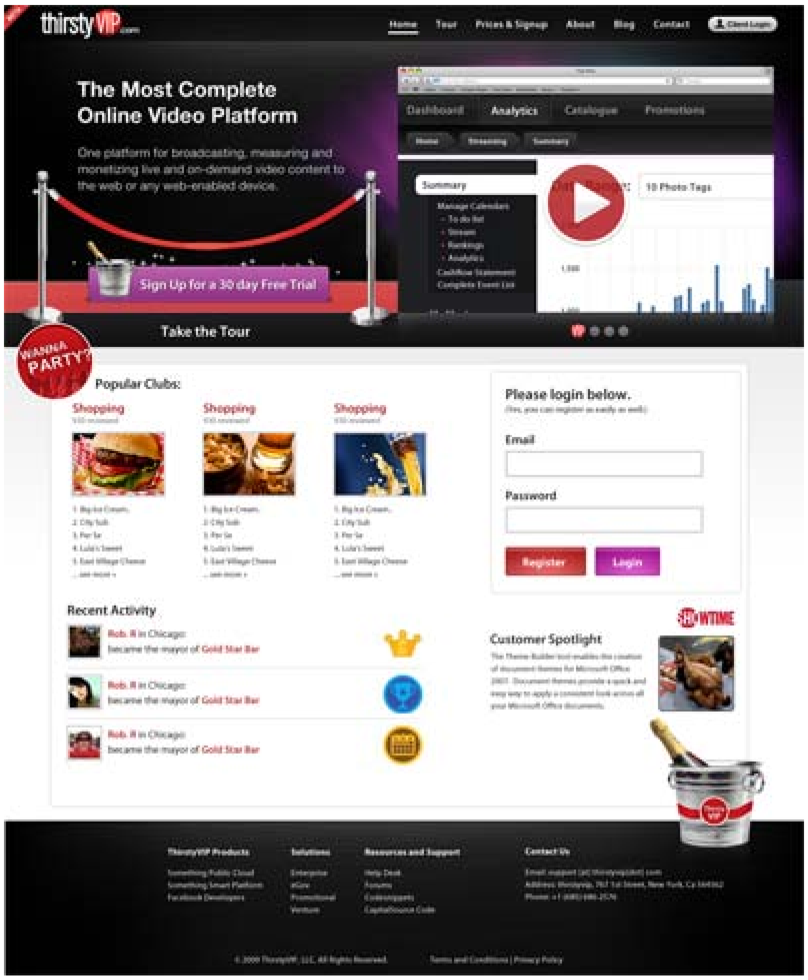
Making these layouts is really an important skill. This tutorial was an overview of the end goal you're looking to achieve here, i.e. a navigatable set of layouts. In the following tutorials we're going to break it down further. Stay tuned.

7-4 SPECCING PRODUCT SPECCING | FINDING INFLUENCES

Ok, so you get the idea that you have to get your layouts very precise and thorough. The next thing to know is that these layouts can't be crappy "wireframes." Checkout what a wireframe is:http://en.wikipedia.org/wiki/Website_wireframe . Lots of companies say they're building wireframes to spec their product. Every time I hear that I laugh! That's the stupidest crap ever. It's not thorough enough. Just look at that image on the right side of that page. Notice the box with the X through it. That's not thorough enough. For the list of attachments, what if the list of attachments doesn't precisely fit in that box? Does the box get taller and now there are 3 boxes on the same row that are not all equal in height and therefore don't look as good as initially imagined in this crappy wireframe? Does it get scrollbars? Is there paging in that box? This wireframe didn't cover that information, and most don't--and that's just the beginning of all the sub-features your application needs to be complete.

Wireframes, whatever they are, do not go deep enough. All they are is just placing little vague components on a page that too generally represent what should go there. What you need to do is graphically design how every page will work, and then pin-point potential problems (e.g. missing paging links) and then design those in, and repeat.

So the technique I've pioneered--and say that because I think have--is copy/pasting elements from other applications/sites and combining them into basically a collage that represents one of the pages I've specced. Let me explain that further: I visit a web application that has a user interface that I like, then i click "print screen" and paste it into graphic design program, and crop out the full-featured component I need and paste it into the current layout I'm working on. That means my "wireframe components" are real components from other sites/apps.



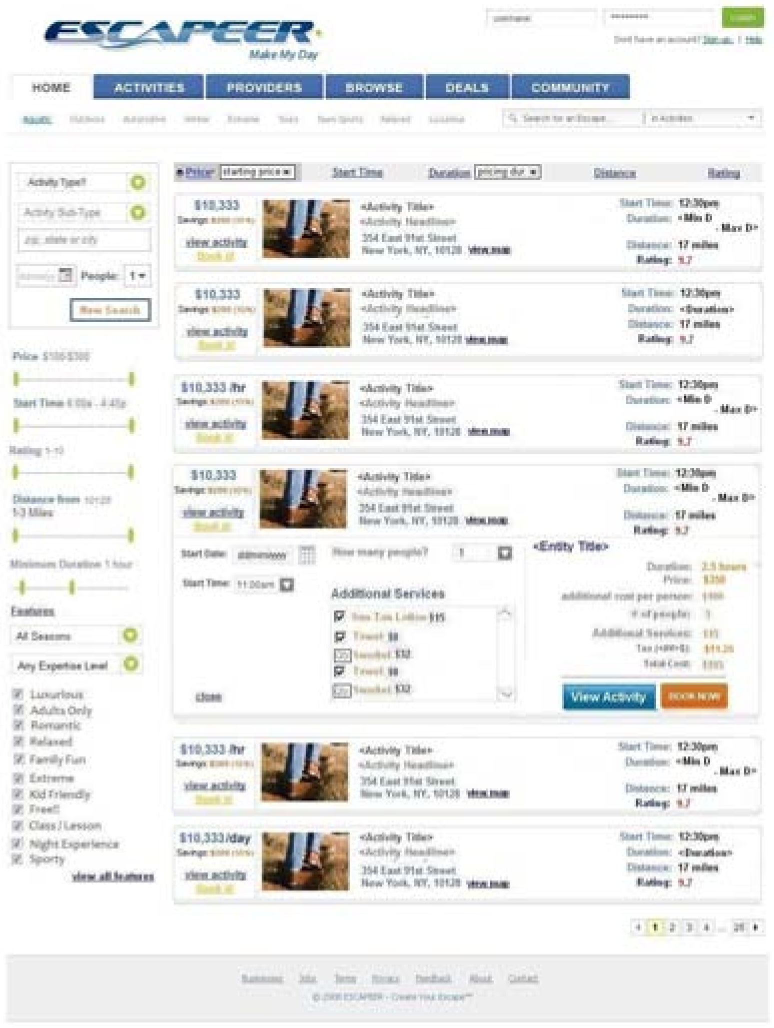
The next thing I do is have our designer start designing what I have. Then when she makes cool components according to the branding style she's been working on--and as I'm continuing to evolve the page--I grab her components and drop them in instead, i.e. replace what I copy/pasted from another site.

The idea is that real graphically designed components will drastically change your view/opinion of how your application should function. You'll learn all you truly need from what other sites/apps have. And then on top of that, the graphic designer needs to get the best influences for what you want as possible. When you're using interface components from other sites/apps, you're inevitably picking components not just for function but because of the beauty of their style. So that means you're also giving hints and clues to influence your designer's direction, and therefore save time and money on the graphic design aspect.

What will ensue is a rinse & repeat process where you're grabbing components from other sites, while your designer is branding them in your app's style (which is also evolving simultaneously), and then you're replacing the components from other sites with the one your designer made. If you're doing this correctly, you'll have magic moments where you completely change one interface component to another one that is a lot more beautiful that accomplishes the same function. The reason this happens is because you'll look at what your designer made and realize it didn't work as well as you thought.

Now imagine you planned your entire application with black and white wireframes, and then you get to the end where you're forcing your designer to design with such tight layout requirements. What will happen then is you'll realize you need to change your whole plan because the ideal interface is something totally different, and in fact causes changes all the way down to the application server code, and of course the clientside Javascript/jQuery code. This is why you need to experiment with what you're application will look like when it's done very early on. It's why you must examine how complete features turned out on other sites, and try to emulate those in all their details. It's why you must try them out in combination with your other components and elements on the page, and see if it works in your case. What worked on another site might not actually end up working on your own site because of everything else going on in the page.

Creators of these other sites/apps ended up solving problems that there is no way for you to know you will have until you get to a phase where you're testing your application. The only other way to think of these things is through your own experience of knowing potential problems from your past completed apps. However, if you're new to this, you won't have that luxury. You know how many times, I've seen page layouts missing paging, filtering and sorting for lists of data! Too many. Those basic things can drastically change the design of the page, and also can definitely change the inner workings of your application code. Filtering data can sometimes depend on complex relationships between the entities in your system. If you're coders don't know you plan to filter the data in certain ways, well, they won't write the code to do so. I need to see a dropdown menu with a list of all the filter options! I.e. just the dropdown with the name of one filtering option in it is not enough. That's how thorough your specs must be.



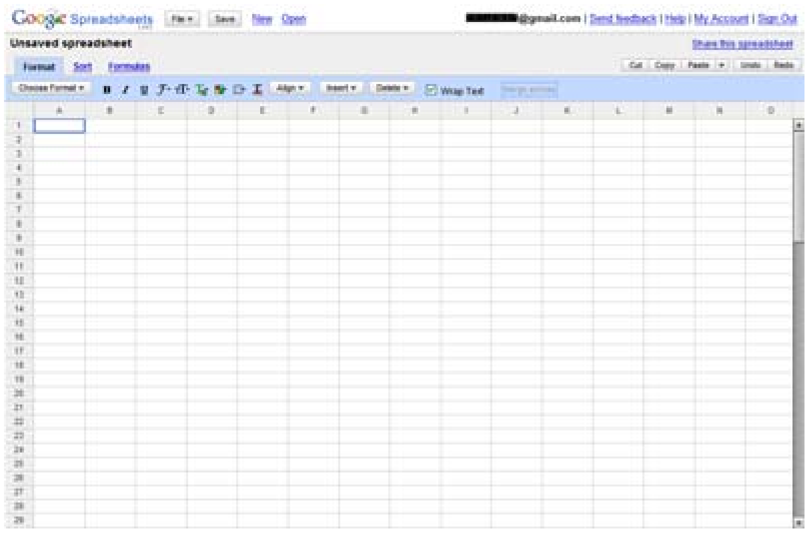
Be thorough in your layouts. Force yourself to imagine farther and farther into the features of your application until you truly can't go any farther. So may say this risks a never-ending stage of "analysis paralysis." For some that will be the case. That will be the case if you're new to this for sure. But if you're new to this, it's only going to get worse once you start development. And what's worse: a costly phase of development that takes forever and possibly never completes? OR, a painfully long phase of planning that doesn't cost you much because you're doing it yourself? I think the answer is pretty clear. If you're serious about your application you'll finish the planning phase.

Honestly, it's taken me years to get as good at speccing as I am. It's taken me executing many products I thought I specced 100%, and realizing I didn't to learn what I need to add to my specs next time. That's just the reality. That's why I recommend doing software as a business (i.e. for clients) before you jump on your own startup, and of course start small when you finally do your own startup. You need to learn from your mistakes. If you're new to this, you'll inevitably make many mistakes. So do all you can on your own dime, and if you're not technical, that means making specs. Send your specs to a professional to review and point out things missing. Hire FaceySpacey simply to do that. Have meetings with your developers along the way about questions they have. Borrow as much as you can from other full-featured launched applications as you can, and play off what they got right. Make these layouts your life. It's your playground. Get quick at re-arranging them. Constantly ask yourself the questions of what's missing and what other features I need to truly make this sing. When you know all the things it needs, then peel out as much as you can that's not crucial, and replace them with the simplest way to patch up the hole. But lastly, if you're new to this: do something so small and get it launched, and see all the work that goes into getting it launched, and by doing so give yourself a dose of humility which will come by seeing all the things you missed in the planning phase that was ultimately needed before it could go live.

7-5 SPECCING PRODUCT SPECCING | ORGANIZED WRITTEN SPECS

Once you have the layouts, you need to write out descriptions for every interface element on each page. This is simple in itself once the layouts are produced. The harder part will be organizing the written specs in a concise non-repetitive format. Generally, the idea is to build a spreadsheet of tasks for each page. Each task is description of one feature of the product, i.e. of one user interface element.

Some times you'll have features that exist on multiple pages, and you may end up describing the feature in the written specs associated with 2 pages. That's fine, but in each place reference the other places. Also have a list of global overarching tasks. If there are things that are similar from page to page--for example how your forms should operate--make a spreadsheet that lists those tasks. If you have a header or sidebar that is always present, give them a sheet of tasks. After that, you should have a list of tasks specific to each page.



The next thing you do is use your project management software to create groups of tasks for these sheets. So that means each page has its own set of tasks listed in your project management software. Then you have your global groups. All this should be very product-centric, i.e. describing user experience, rather than deep technical notes.

Though, in my opinion, don't be too strict with yourself. I you are technical, and have some technical notes to add, add them. However later, your tech team will add deeper technical tasks as sub-tasks. Therefore you need to be using project management software that allows deep nesting of sub-tasks. We use FogBugz, which we'll cover in the next tutorial.

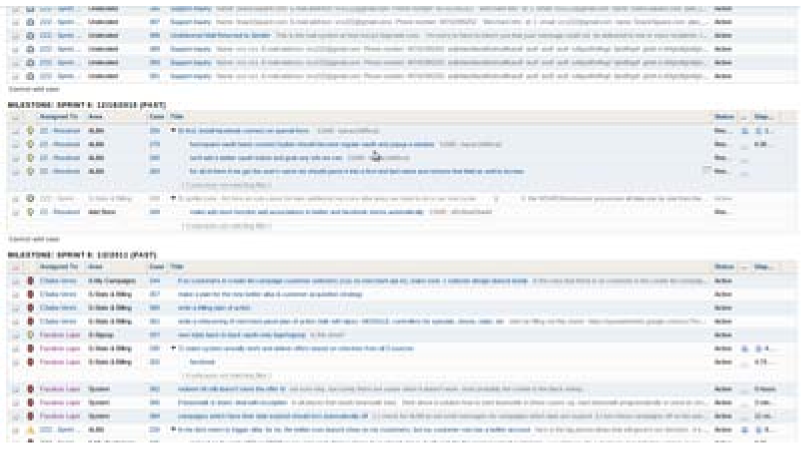
7-6 SPECCING PRODUCT SPECCING | TASK ORGANIZATION & BUGTRACKING (Fogbugz)

As mentioned in the previous tutorials, I suggested building spreadsheets of your tasks. Specifically, I'm talking about using shared google spreadsheets. Build one spreadsheet, and add sub-sheets to it. Don't juggle multiple spreadsheets use URLs everyone in the team has to remember. Start your project out with one spreadsheet with sub-sheets. Make the first sheet that appears in it something general, like directions of how the whole sheet is organized, for example. Also, make the spreadsheet accessible via a public URL. Don't do the whole "share" thing where you invite people. People have different gmail accounts and google apps accounts, and won't be able to visit the link, and will therefore visit the link less (i.e. when they're logged into the gmail account associated with it) and in some cases never. In general, when building software, reduce as much friction as possible between accessing available documents, code, etc. And if you have multiple documents, link to them all from the first page of the master spreadsheet and tell everyone just to bookmark that sheet.

Ok, so you started with spreadsheets. I love spreadsheets because they're very malleable. You can add new columns for new data, new sub-sheets, etc. People can write comments in the rows, and it's a great place to collaborate with very little friction.

Once you're done however, you need to transfer these tasks to your project management software.

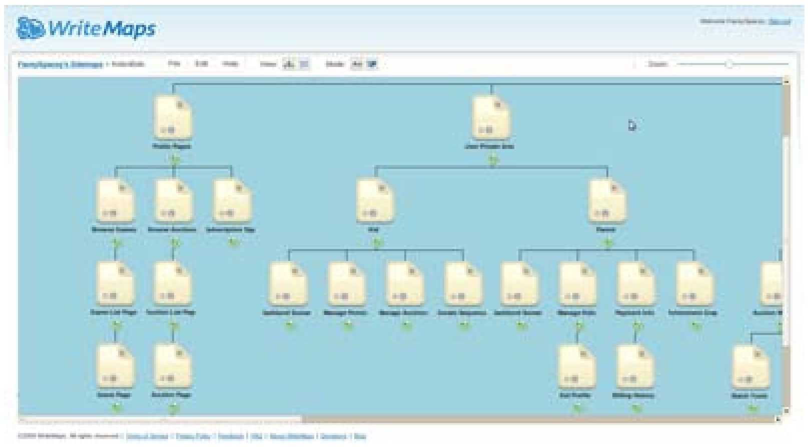
So on a per page basis (and per state basis), we create written specs closely attached to the layouts, and organize these feature requests in FogBugz, Joel Spolsky's project management software. Joel Spolsky FYI is the creator of Stack Overflow, and a famous developer from Microsoft who worked on the Excel project. Anyway, the key here is that the list of tasks are coupled to their layouts, and then that FogBugz is arranged in a way to mirror this association just like the sub-sheets represented one page, or similar global tasks. Sometimes you'll have group/sheet of tasks just for one state in a page if there's a lot of interactivity going on.



Other shops may do more of a Bottom Up approach, where specs are written in association with the underlying data structures. That's fine, but not yet. We find the problem with starting with that approach in $50-150k contract projects is that our clients can't closely associate the tasks being executed to precise features you're looking to see. If you're small startup, even working internally with an inhouse team, you'll be in similar boat. For example, a database structure may be prepared and it took the developers a week to do, but you, as an end user testing the system, will have no way to associate the completed tasks to completed features you can use, and then you have no idea what you're developers were doing during that time. In other words, you must optimize your process so at all times you know what's going on in the form of testable features. Later in the technical speccing tutorials, I'll describe how you're developers should pin-point technical spikes and create the tasks according to a more bottom up approach.

So back to organizing Fogbugz. In Fogbugz, you will create what's called an "area" and we use each area to group all the tasks in one of those sheets, i.e. that mainly correspond to pages at this point. You'll then create a "filter" that will group tasks by Area first. This will create neat groups on the page obviously. Later on you'll also arrange the same set of tasks in a different "filter" that groups by open, resolved and closed tasks. You will do so on a sprint by sprint basis. We'll cover this more in later in theSprint Planning tutorial. The overarching idea is that Fogbugz allows you to create multiple views through which you can look at your data, and again these are called "filters." For now we're creating a simple filter to view all your tasks nicely grouped similar to your spreadsheet.

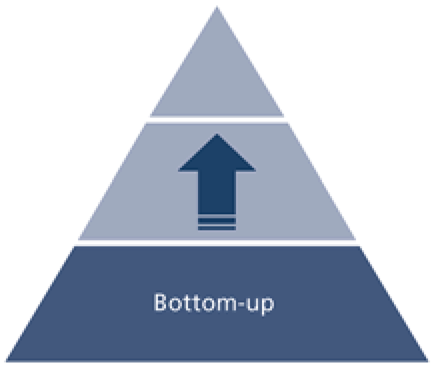
The next thing you will do is organize the columns in the filter so that only the columns you need show. These columns are the title and description and nothing else. You want to be able to look at just the data you need to know now about the product now, not how the execution of the product is evolving with columns for things like status, etc. Later on you may add "status" back to it, but this is your master "product backlog," as they'd call it in Scrum (http://en.wikipedia.org/wiki/Scrum_(development)) , which we won't cover now because we're focusing on our own precise techniques, rather than the history of how we've borrowed stuff from other methodologies.



The last thing you want to do, which may actually deserve it’s own chapter, is you want to create a sitemap of all your pages. We use: http://writemaps.com/ . That tool will let you build a tree style sitemap and then link out to pages on the web. So what we do at FaceySpacey is upload all the layouts to the web and link to these layouts from the sitemap itself. This way you can quickly visualize the application as a whole via the sitemap, and dive deeper to see each layout if you want. Then the last touch is in each sitemap icon we also put in the link to a fogbugz filter that filters tasks just down to the tasks in the area corresponding to that page. This way you can quickly navigate from the sitemap to an individual page layout and its corresponding product tasks.

7-7 SPECCING TECH SPECCING | OVERVIEW

Just because we focus on a Top Down approach, where the specs and “product” lead the way, does not mean we also don't take a Bottom Up approach--which we do simultaneously. The difference is that before each decision we make for the underlying architecture, we have a precise product goal in mind we’re working to achieve. A lot of developers may jump to modeling database tables and columns, and they’re corresponding PHP classes. They’ll imagine in their head features, and instead of making layouts as a way to embody them, they’ll model their ideas at the architectural level. However, we find that there is too much learn through exploring your products through layouts that this is not the right approach. You may end up with a totally different product, as previously mentioned, if you take the time to imagine it through layouts. Hopefully you will because there are so many options of what to build, and you need to be sure you're building the right product if you're going to follow through with it over many months while possibly spending lots of money. There's so many factors to consider, and the best way to consider them is by looking at what you're end users and customers will be looking at.



So in this part of our speccing process, we'll come up with the entire database design. Then we'll prepare empty code files for all the code we anticipate building. Lastly, we'll pinpoint potentially problematic areas in the application, and discuss and write a plan for how we will address them when we get to them. This way the entire plan isn't thrown off track when we find a major challenge deep in the app that changes the way tons of other stuff should have been coded, or even the way other features should have been.

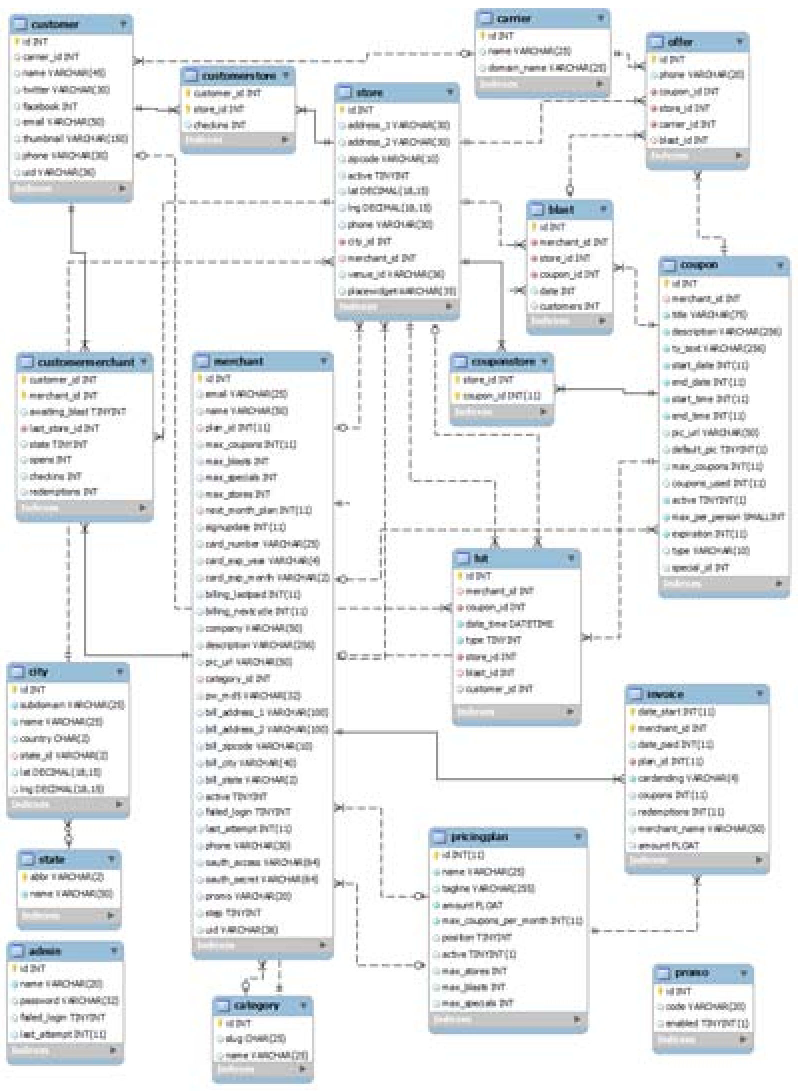
One key thing here is that you don't say to yourself that the product speccing stage is over and now it's time for the tech speccing stage. While you should do a ton of the product speccing stage first, there should be a lot of overlap in the middle, and potentially the product speccing stage will go until the tech speccing stage is done and they'll end at the same time. The idea is that while you're examining the technical implications of the product you've planned you'll most likely find major pitfalls that will cause major time-sinks in development time. Finding them is crucial if you have any hope of finishing your product. Plan to find out that at least one thing is too technically complex to develop that you have to go back and drastically change your product. This will happen, and if you suck it up and make the product changes, you'll save yourself a lot of pain that would come later--even if it means you have to go back to the drawing board on layouts you have to craft and have to pay your designer to redesign the new stuff you came up with.

7-8 SPECCING TECH SPECCING | DATABASE DESIGN

So what does the tech speccing itself look like more precisely? Well, first it's all about designing your database. To do so, you need to find all the entities in your product, i.e. stores, users, videos, articles, coupons, campaigns, status updates, checkins, the relationship between these entities and each other, and build database tables to store their corresponding data. Then you add columns to these tables to hold that data. Modeling a database for these types of Web 2.0 applications really isn't that difficult, especially if you're product is specced out well before hand.

The hard part will be pinpointing special relationships between your main tables, and creating additional tables that don't have clear names like "store" or "user." For example, if you want to list all the users that edit an article, you'll need an "edit" table that associates a user row with an article row and has a column for the date/time of the edit. It's not a crazy concept to experienced developers by any means, but the point is that there will be harder to pinpoint "junction" tables like this. If you've kept your product to a minimum there will be very few of these. One way to know you're doing too much is if you have too many junction tables. Junction tables fyi typically represent "many to many" relationships, i.e. where an article can have many users that edit it, and each user can edit many articles. For example with SnackSquare, we had junction tables like "CouponStore," "CustomerMerchant," "CustomerStore," etc. There is ways to get around needing these junction tables. The main one is to plan a simpler product that doesn't need to present every single metric to the end user. Junction tables often embody extra bonus info. Forget that bonus info for your initial launch.

The final thing you should do is generate a diagram of your database. You can easily generate a diagram of your MySQL tables using MySQL Workbench. Here's an example:



7-9 SPECCING TECH SPECCING | APPLICATION ARCHITECTURE

After you've built your database tables, in Yii (our PHP Framework of choice) you can actually just generate PHP model classes (and controllers and views) that represents that same data, but obviously in PHP.

If you're reading this as a non-technical reader, this may get too advanced for you, but I'll give you a quick overview. In the sort of applications these tutorials are written for (and the sort of application we build at FaceySpacey), the code is written according to an MVC ("Model View Controller") architecture. The Yii Framework is based on this extremely popular pattern. The core concept to at least have a cursory understanding of here is that "models" represent your data in your database, "views" the templates that your web pages are based on, and "controllers" connect the whole thing together, making use of your models and triggering the display of models based on which URL your users visit.

Typically we won't generate the controllers and views though as we have our own way of organizing them that's a little against the Yii way, which we find allows us more flexibility and is standardized within our company.

Building/planning the technical architecture doesn't stop however with the generation of some near empty shell model classes. What we'll do next is group the layouts by controller and determine which pages our controllers will generate. The controllers generated by Yii's CRUD ("CREATE, READ, UPDATE, DELETE") generation tools will make a controller for each model. Often your app won't need to allow users to create/edit/etc every single model (i.e. database table) you have in your system. For example, the "edit" table described in the previous tutorial doesn't need a way for users to create, update or delete them. So for the models/tables that do have corresponding interfaces, we will however follow that Yii convention. For all the other pages, we'll add them as empty shell “action methods” to a controller, whose name will logically represent all those pages. If you aren’t technical, this aspect of the speccing tutorial is probably going over your head. Generally speaking, the idea is just to assign the pages that you’re app will generate to specific code files, which technically in this case are called “controllers.”

We’ll then do the same for views. Again, we won’t follow the Yii way precisely. We’ll use their theming features and create all the views in a Theme, and organize the view files in a way that makes sense for them unrelated to the controller action methods that will trigger them. We’ll organize them in a way that a designer would think makes sense--since a designer will continue to work on them throughout the rest of the project. That way is specifically where similar layouts are grouped in the same folders, even if the layouts are from different areas of the application triggered by different controllers. We just find that not forcing your “view” files to match controllers makes it really easy to browse view files and jump between editing them. You do have to kind of switch gears a little more than if your views precisely paralleled your controllers, but we find that switch easy to make.

So either way, the idea up until now is we create basically empty files, title them properly, title some of the methods they contain, and arrange them according to a folder structure we plan to keep for a very long time.

The next bit is we’ll pinpoint problem areas, usually areas that require using 3rd party APIs and libraries, and stock the system with these tools so it’s there waiting for us. If we can, we’ll imagine our own client classes that use these tools, and again create and title empty files to represent them and put them in the appropriate place. Often when coding, you’ll be tempted to just stuff your models full of methods--right now is the time where we figure out our own components to build that model class methods can be clients of. So we'll stuff our "components" and "extensions" folders with shell classes, just sitting there waiting to be flushed out as a reminder of what we need to build.

In short, your project's file system becomes a to do list of what you have to code. That's what this part of the tech speccing phase is all about. What this does is help developers see where everything is going to go. There are lots of decisions to be made here--for example if you're going to bundle up part of your application into what's called a "module" or a "widget." So just the existence of these files are notes to developers saying how something should be coded.

Next we'll set the configuration settings of the application so that all the 3rd party libraries are included, URLs are formatted in a beautiful way that will be consistent from the start, so the database is connected to the application, etc, etc. And then we'll test developing a few things to make sure it all works as expected. What that means is we'll make the application generate boring pages where all parts are used, i.e. the controllers, views and models, and make sure everything is working as expected.

If there are some particularly scary parts, developers should start working on them asap before all the final decisions for the product are sealed to make sure you will be able to do it. Basically send out some scouts into the most complex areas of the application and have them work on it as soon as you can. As I said before, there will be overlap between the product and tech speccing stage. As soon as the tech speccing stage is far enough long that you can start building the file system just described, get some developers experimenting with coding the hardest parts to make sure you've made some of the right decisions. You'll inevitably find that some things take so much time to code properly that it will be a big time saver to back-track and change the product (and redesign it).

The final aspect in this phase is that we'll create sub-tasks to the product tasks in Fogbugz. Obviously these sub-tasks (and often sub-sub tasks and so on) are of a technical nature. If there are overarching technical tasks that don't apply to a specific product area (i.e. page) or perhaps apply to many pages, we'll create "areas" for these "technical spikes." This whole phase was about finding the problem areas in the technology. So when we find them, we need to document them not just through the organization of the file system, but as Fogbugz cases/tasks. In short, we'll add all the technical tasks we can to the product tasks in fogbugz, and intertwine them in a logical way. If we can associate them directly with product tasks and pages, we will, and if not we'll create new areas that group a set of related technical tasks.

7-10 SPECCING TECH SPECCING | SPRINT PLANNING (And More on Fogbugz)

As you may recall from the Fogbugz chapter, I mentioned creating "filters" to view your tasks per sprint. This tutorial is all about those sprints. A sprint is a set of tasks you plan to get done in a given period of time. The goal behind sprints is to get to a point where each week you accomplish what you planned. That said, you will overestimate what you can get done in the beginning. That's fine. Move the remaining tasks to the next week's sprint, maybe add a few new tasks, but make the sprint size smaller than the previous week's. Sprints can be of any length. Ours are one week at a time, and that's just how we recommend you do it. For larger applications, such as video games, a longer sprint length may make sense, but that's not the sort of application these tutorials are written for.

Let's talk about the value of sprints. Sprints establish predictable rhythm of execution through which you know when to expect results. The type of applications we do (to begin with at least) are medium-small applications that capture the precise minimal viable product ("MVP") you need. Therefore, we can do our sprints in increments of a week at a time, rather than larger increments common to desktop applications of yesteryear. This insures that non-technical stakeholders feel the pace at which developers are going from the start. This allows everyone to gauge progress, and accordingly feel comfortable with the process every step of the way.

At FaceySpacey, starting on Thursday until Monday morning, our clients in conjunction with our dedicated testers test the application. By the way, teams without dedicated testers are wasting your money by having highly expensive engineers be solely responsible for testing, and waste your time by putting the responsibility completely on you, somebody just getting acquainted with the process of software development. The final thing we do here is we do not move to new features until the current features work perfectly. The reason is simple: fixing bugs later when coders are less familiar with the code at hand takes many times longer to fix than fixing it right when they first made it. Therefore, if you don't complete tasks in the current sprint, find new related tasks, or find lots of bugs, those tasks should go into the next sprint before adding new distinct features.

Back go Fogbugz. In Fogbugz you will take all the tasks from the "product backlog" previously described and assign them to what Fogbugz calls "milestones," which we use to represent sprints. Basically you'll assign the first tasks you think should be done to the first milestone/sprint, and then the next set of tasks to the second sprint, and so on. You'll do that until all the tasks in the entire product backlog are assigned to milestones. Make sure to note that most likely you'll end up with more sprints than you originally planned as things take longer than you initially thought, and you find your developers working on a task in a following sprint that was planned to be done in the previous one. That's fine. What we're talking about here is getting an initial birds-eye-overview of what will follow in the development process. Specifically, you're pinpointing the order with which you should do things. That order will change, and you'll alter your sprints as you go, but give yourself and your team an idea of how you imagine things going from the start.

When you build these sprint filters that lists all the Fogbugz tasks (which by the way are called “cases” in Fogbugz), you’ll group the tasks by OPEN, RESOLVED or CLOSED rather than by Area. These are called “statuses” fyi. Throughout the week you’ll see tasks move between these 3 groups from open, to resolved to closed. In Fogbugz, this is like a vertical Kanban chart. If you don’t know what a Kanban chart is, check this out: http://www.infoq.com/articles/agile-kanban-boards . It’s a way to visualize tasks go through their sequential stages from being started to completed. At FaceySpacey we find this to be a key motivational tool and tool to help us focus.

Back to how to build these sprint filters. Now, since tasks aren’t grouped by area anymore, you will need to add a column with the area so you know what part of the application the task has to do with. If your team is using the time tracking tools (which are used to generate graphs that predict when things will be completed), you’ll also want to add a column that states the time elapsed on the task. This requires that your developers record how long they work on each task. It takes a lot of discipline within a team to do this. Luckily since you’ve been using the sub-tasking feature previously mentioned, a developer can simply state how long the overall task took, like at the end of the week. Fogbugz has great algorithms to make estimates on when things will be done based on past performance. So any data here provided your developers is better than none and enough for Fogbugz to produce some pretty--and useful--graphs.

The last 2 columns to add to this filter are one for which developer each task is assigned to, and an order column provided by the “backlog” plugin. Basically you can set the order developers should do each task.

Fogbugz offers a lot of functionality, but I find the most important functionality is just how flexible its filter system is in how it allows you to re-organize the same tasks in as many different views as you want. For example, I often list tasks in a filter that groups them by developer so I can quickly see what each developer has.

Another trick I intentionally didn’t mention above is I actually create fake users for “RESOLVED” and “SPRINT BACKLOG” in order to make the vertical Kanban chart. The reason is this: the “OPEN” status is embodied by being assigned to a developer. So therefore the groups in the filter described above have the names of developers at the top, and then when each developer resolves a task the task all goes to the same “RESOLVED” group. Before a developer starts working on a task, the task is stored in the “SPRINT BACKLOG” group. And of course after the task/case is tested, it’s sent to the CLOSED group. So the trick here is that we’re user real and fake users to make the groups, and building the filter so that users are chosen for how to group the tasks. The result is that throughout the week you can watch the tasks move from the top of the page to the bottom as they go from each group to the next, and at the top there is little mini groups for each specific developer, rather than all the tasks grouped together. You could add a “developer” column to the task list, but I find visually this grouping very effective.

Conclusion & Further Reading

We at FaceySpacey hope you've enjoyed our FaceySpacey Bible, and are coming away many times more ready to succeed at your next software startup. At the very least, you should have a birds-eye-view of what you need to do to get your startup, and have quelled a lot of insecurities you may have had regarding how you should execute it. That said, I will point to you to what you should do next.

As promised, the following is a list of the precise books I read to master web development using HTML/CSS, Javascript, PHP & MySQL. They are presented in the best order to most efficiently learn the subject at hand. It's similar to the order I read them in, but enhanced based on what I learned and the order I wish I read them in. Good luck:

HTML/CSS:

CSS Mastery: Advanced Web Standards Solutions

http://www.amazon.com/CSS-Mastery-Advanced-Standards-Solutions/dp/1430223979/

Before you start coding PHP, Javascript, etc, understand how HTML works. This is where you start. HTML is easy. Read this book in combination with studying the HTML & CSS tutorials on w3schools.

PHP & MySQL:

PHP & MySQL For Dummies, 4th Edition

http://www.amazon.com/PHP-MySQL-Dummies-Janet-Valade/dp/0470527587

This book--well an older edition--I read a year before I got serious about learning to code. I read it and didn't actually code anything i learned, but what it did was plant seeds in my head with regards to what programming is all about and what databases are all about, and how to connect the two. It assumes very little in what you may already know, and is an excellent start in your journey to becoming a master programmer.

PHP Object-Oriented Solutions

http://www.amazon.com/PHP-Object-Oriented-Solutions-David-Powers/dp/1430210117

This book is where I learned what OOP is. I didn't get the hang of it until reading the following book. Don't worry if you read this and have a hard time with it. This book and the next each have introductory chapters that go over how OOP works. It took me reading basically this book and the next book about the same stuff to get it. This book is a lot less complicated than the following and dives into practical examples & problems, whereas the next is a lot more theoretical.

PHP Objects, Patterns and Practice

http://www.amazon.com/Objects-Patterns-Practice-Experts-Source/dp/143022925X

After reading this book, I basically mastered OOP. It's a very hard book to get through if you're new to OOP, and goes into some very advanced stuff, specifically tons and tons of "design patterns." The design patterns are presented in as basic of a form as possible, but they weren't very practical like examples from the previous book in that you probably will never actually need any of the code used in the book. Either way, this is my favorite Programming of all time because it taught me how to think like a coder and how to solve complex problems with concise refactored solutions. It really showed me what is possible with OOP. You don't know PHP unless you've read this book.

Pro PHP: Patterns, Frameworks, Testing and More

http://www.amazon.com/Pro-PHP-Patterns-Frameworks-Testing/dp/1590598199

I read this book too just to solidify my experience with PHP, and cover all my bases. Check it out. It's optional.

PHP Functions Essential Reference

http://www.amazon.com/Functions-Essential-Reference-Torben-Wilson/dp/073570970X

At some point during my study of PHP I found this book and decided just to learn every PHP function available so that I could better understand the examples in the above books. Start reading this early on, and complete the whole thing. You'll quickly learn patterns in how PHP functions are named, and as a result be able to guess what a function does within the context of the examples in the above books--even if you don't remember precisely what it does.

Agile Web Application Development with Yii 1.1 and PHP5

http://www.amazon.com/Agile-Web-Application-Development-PHP5/dp/1847199585

I read this book in combination with reading the Blog Tutorial and Definitive Guide on YiiFramework.com. When you're done studying all these materials, you'll be amazed with how much power you have. This book isn't hard to read either. You'll love it if you reach this stage!

Javascript & jQuery:

Learning jQuery, Third Edition

http://www.amazon.com/Learning-jQuery-Third-Jonathan-Chaffer/dp/1849516545

jQuery is a framework built on top of the native browser language of Javascript. Usually one would recommend you learn the base language--Javascript--before learning an abstracted framework on top of it--jQuery. However because of the nature of jQuery and how comprehensive it is and because of how quirky Javascript is coming from PHP, I found it best to jump to jQuery and immediately start accomplishing the DOM manipulation tasks I needed. And ultimately because of syntax similarities between PHP and Javascript I was able to get productive in Javascript without studying a single book just on Javascript. One thing I did different when studying this book from the PHP books is I did every single tutorial as I read it. The reason is because when I learned PHP, I was learning my first real programming language--so it took me a lot of time to just digest things before I could code a single line, which is why I just read PHP book after PHP book before I got started until it all made sense. However, by the time I got to Javascript & jQuery, I understood how programming in general works and found it helpful for memorization purposes to immediately start doing the tutorials.

jQuery 1.3 with PHP

http://www.amazon.com/jQuery-1-3-PHP-Kae-Verens/dp/1847196985

With this book I didn't do all the tutorials like I did with *Learning jQuery*, but what reading this book did for me is taught me precisely how Ajax works and what it's all about. After reading it, coding features that required Ajax using Yii and PHP was obvious and a no-brainer.

JavaScript: The Good Parts

http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742

This book gave me a deep understanding of the Javascript language and what it's truly all about. After reading it, many hours of debugging and head-scratching when coding Javascript & jQuery were removed from my schedule--because I finally learned the quirks of the Javascript language I needed to know.

Pro JavaScript Design Patterns

http://www.amazon.com/JavaScript-Design-Patterns-Recipes-Problem-Solution/dp/159059908X

Now this book took my Javascript game to the next level, gave me an idea of how jQuery was built, taught me how to do things similar to how you would in a "classical" OOP language like PHP, and completely ended any remaining head-scratching I was having with Javascript, particularly with how "scope" works in Javascript.

Linux:

The Official Ubuntu Server Book, 2nd Edition

http://www.amazon.com/Official-Ubuntu-Server-Book-2nd/dp/0137081332

Note: by the time I read this book I had already learned Linux through blogs on the internet. The best thing I can recommend you do is install Linux on your computer from the Ubuntu website, and start navigating around the command line, practicing Linux commands you learn off the web. Just google "linux tutorials" and you'll be off to a running start. That said, by the time I got proficient in Linux and after I read this book, I felt confident that I really knew what I was doing and had practical solutions for the most common problems you'll face at the command line.

Apache Cookbook: Solutions and Examples for Apache Administrators

http://www.amazon.com/Apache-Cookbook-Solutions-Examples-Administrators/dp/0596529945

This book I treat like a pocket reference and still refer to it often since it's impossible to remember all the different Apache configurations, given how comprehensive this web server application is. I did read it through when I first got it. I kinda skimmed it though--just to get an idea of what is possible. Getting an idea of what is possible without mastering a subject matter is so important in programming because you'll know where to look when you face a challenge that the subject matter can solve.

Pro Bash Programming

http://www.amazon.com/Bash-Programming-Experts-Voice-Linux/dp/1430219971

This is a little advanced for readers of the *FaceySpacey Bible*, but I'm putting it here because it really took my Linux skills to the next level.

NON-TECHNICAL BOOKS:

Smart and Gets Things Done: Joel Spolsky's Concise Guide to Finding the Best Technical Talent

http://www.amazon.com/Smart-Gets-Things-Done-Technical/dp/1590598385

If you plan to grow a small application into a large company, this book is a must. It's short. Read it.

SEO Book.com

http://www.seobook.com

This obviously isn't a book, but I read their entire site like a book, and its creator, Aaron Wall, expects you to read it like a book. When I was done reading it, I felt I was completely up to speed regarding what SEO is, how search engines work, and practical techniques to get better rankings in search engines.

For daily Startup Wisdom, checkout FaceySpacey.com/blog daily. And don't forget to download the entire No Bullshit FaceySpacey Bible or more individual chapters here: http://www.faceyspacey.com/resources?section=book .

*Thanks again from FaceySpacey and be sure to check out **FaceySpacey.com** often for further knowledge we kick to take your Startup to the Stars!*