Building Software Products in a Weekend

by Matt Kremer

Building Software Products in a Weekend

by Matt Kremer

A special thanks goes out to my wife, Sarah Beth, and my baby, Oakley, for letting me do crazy things like write eBooks, build software products, blog, and turn our basement bathroom into a podcasting studio. I love you guys.

I'd also like to thank the Entreprogrammers: Derick Bailey, John Sonmez and Josh Earl. They've helped me figure out what path to take and have made my commute suck a lot less with their podcast.

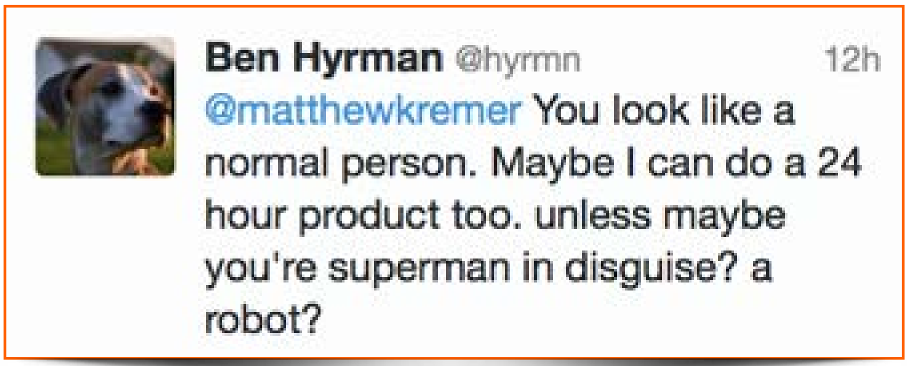
Table of Contents

Introduction

Hey, I'm Matt Kremer.

Before we dive into the methods and strategies I use to build products in hours instead of weeks, I wanted to tell you a little bit about myself and why this book was born.

While writing the first version of this book for a 24 Hour Product Challenge (please send your feedback to matt@ mattkremer.com!) someone reached out to me on Twitter with the following:



Ben Hyrman @hyrmn 12h
@matthewkremer You look like a normal person. Maybe I can do a 24 hour product too. unless maybe you're superman in disguise? a robot?

So let's start off with a few things about me:

- **I am just a normal guy, not a robot or Superman. Although that would be pretty sweet.**
- **I've got a wife, a 10-month-old son, two dogs, a lizard and a full-time 9 to 5 day job.**

Really. I'm just a regular dude who really likes marketing, business and programming...a lot. I tend to work on my blog during my lunch breaks, and I run "The MattKremer.com Podcast" whenever I have time and a cool guest to sit down and chat with.

One year ago, almost exactly, I had the idea for Kobra.io, a real-time collaborative code editor with built in video and voice chat.

I began slaving away, putting all of my free time into the project for a period of six weeks. I'm pretty sure my wife hated me during that time period, and for that, I apologize.

I kept telling myself, "Just one more feature, then I'll launch!" And finally I did, to an email list of over 2000 people that I had collected.

But, it turns out no one really wanted what I had made. I had wasted six-weeks of my life, my wife was mad, and I was distraught. I completely abandoned the project.

A few months ago, I began researching any books, blog posts and podcasts I could get my hands on regarding building and launching products. I decided that I was going to re-launch Kobra.io, but this time I was going to make a Minimum Viable Product.

I slimmed down the feature set and held a pre-launch sale that brought in $2200 in four days. Not a fortune (like I said, I'm a normal guy), but not bad if you ask me!

Then I spent a whopping **thirty-six hours** programming the new version, and released it to the public. Yes, you heard me right. I programmed the new version in a weekend.

Throughout this book I'm going to teach you the methods that I use to make sure I'm not wasting any time so that I can build Minimum Viable Products as fast as humanly possible.

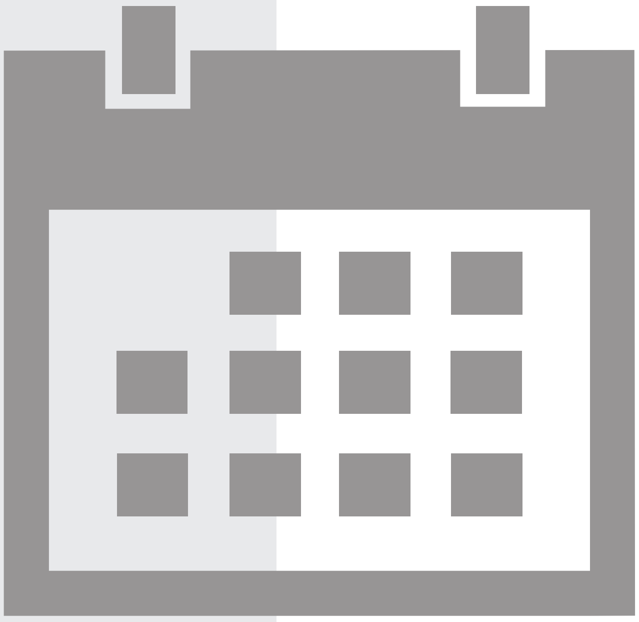
Why would I want to build a product in a single weekend?

Well, perhaps you're like me. You've got a family and a full-time job, but you still want to build an idea that's been bouncing around in your mind.

Or maybe you are preparing to compete in a Hack-a-thon or Startup Weekend event where every minute matters. I can tell you that at the last Startup Weekend event I participated in, my team was one of TWO teams that had actually programmed something, and we took first place using the methods in this book.

Perhaps you just want to test your idea before going full-speed ahead (more on this in the next chapter).

Whatever the reason, it can definitely be an exhilarating experience. So maybe you just want to try it out for fun, or for a challenge!



About this Book

This book is not meant to be the end-all-be-all book of building products. I'm not going to talk about how I met some famous programmer or interviewed the top startups to develop a framework specifically built to speed up development time.

If you want all of that "famous-person" fluff, I recommend you stop reading and go find a different book.

The examples that you'll find throughout this book are all things that happened to me throughout my development career or that I experienced while building Kobra.

The current version of this book is also not a long drawn-out full-size book. Instead, it's a small easy-to-read primer on a new method of development that I find works great for me. I hope it will work great for you too!

How to Reach Out to Me

I'm always willing to help whenever I can, so feel free to reach out to me if you ever need anything. Also, if you find any typos in this book or a section just doesn't seem to make much sense to you, I'd like to know!

- **My blog: https://MattKremer.com**
- **My Podcast: https://MattKremer.com/podcast/**
- **Email me: https://MattKremer.com/about-me/**
- **Twitter: @matthewkremer**

Stay up to date with my blog posts and other great information (like updates to this book) by making sure you are subscribed to my email newsletter:

https://mattkremer.com/newsletter/

Thanks for checking out my book, and I hope you enjoy it!

Matt Kremer

What is a Minimum Viable Product?

One of the most important things you can do when trying to build a product quickly is make sure that your idea isn't too big but that it's also not too small.

Sound easy? It's not. Trust me.

Chances are the first time you try to create a product you're going to try to build something so big that it never actually gets completed.

Or maybe that's already happened to you. If so, I highly recommend utilizing the methods in this book to start over and make your idea smaller.

Why build a "Minimum Viable Product?"

MVPs serve one very specific purpose: they allow you to determine whether or not your idea is a "good one" with the smallest amount of effort on your part.

Would you rather find out that no one is going to use your product in twelve months? Over the course of a few weeks? Or maybe even in a single weekend?

The goal is to **test your product** before putting a large amount of time, effort, resources and dollars into it. If you find that a good amount of people use or pay for your "Minimum Viable Product" chances are a good amount of people will use or pay for your "Giant Perfect Product."

So why waste time creating a "Giant Perfect Product" when you can just dip your feet in the water and build a "Minimum Viable Product?"

The term "Minimum Viable Product" has been around for a while now, but it tends to be misunderstood. So I want to go through each term separately, and then we'll bring them together at the end of this chapter.

The "Product"

We're going to start with the easiest term to understand: product. This is how many startups or side-projects begin. You have an idea for a product and you know how it will help your customers or users.

One of the scariest things I hear from people that are still coming up with their ideas, however, is that they haven't done any competitive research.

In fact, one of the easiest ways you can save time and money before launching your product is by making sure that competition does indeed exist.

By confirming that they exist, you are further validating that there is potentially a market for your idea.

In order to better define your product, I recommend that you try to figure out who your competition is. Buy their products and research them.

If you can, try to determine if they are already profitable instead of hemorrhaging money. If they aren't profitable, that might actually be a bad sign. It might mean they can't find enough customers, they aren't charging the right amount of money, etc.

What can you do better? Are there parts of the competing products that are painful? How can you reduce this pain?

Alright, so you've got your idea. You've found your competition and done some research. Just build the fifty features on your to-do list and you're good right? Not necessarily.

What happens if you were wrong? What happens if no one actually wants your product?

That's where the other two terms come in. Both "minimum" and "viable" are there to "rein you in." To make sure that you're not building too much, but also make sure that you're building enough to get users.

The "Minimum"

The worst thing that you can do when building a product is "make it perfect."

Chances are, the more "perfect" your product is the more time you've spent on it and the more features you've loaded it with. But a "perfect" product does not guarantee success.

Instead of launching to applause, you may actually launch to crickets.

The first version of Kobra was built over a period of six weeks. During those six weeks I kept adding feature after feature. Every time I would tell myself "No one is going to use Kobra unless I have X in it."

Finally I pulled the trigger and launched it. I had an email list of two thousand people waiting. Guess how many people were using it after three months.

Eight. Eight people. My "perfect product" that was loaded with features was obviously NOT perfect.

Instead of hiding in the shadows until your product is perfect, it is much better to start small and make improvements frequently. Base these improvements on real customer feedback instead of the “ideal customer” that you have in your head, because the “ideal customer” might not actually think the way you assume they do.

The truth of the matter is that everyone thinks their product is going to succeed; otherwise they wouldn’t be building their product in the first place.

Gradually improving your product and getting input from users as soon as possible is what is going to allow you to succeed in the long term.

If there is a feature that you think isn’t necessary, don’t be afraid to put it on the bench for later. After you have users on your product, you can reach out to them and ask: “Is this a feature that you would like to see?” or “Is there anything that you think is missing from our product?”

I encourage you to go through every feature on your desired feature list and ask yourself, “Is this feature TRULY necessary for launch? Or can it wait until later?”

For Kobra, I knew I needed very few things to test it out on its second launch:

- **The ability to code collaboratively.**
- **Video & Voice chat**
- **Text Chat**
- **Common editor settings and coding languages that people are used to.**
- **A payment system.**
- **The ability to log in and log out.**

I didn’t even program a “user dashboard” to show users what files they had created. You had to manage your unique URLs yourself and share them.

After I launched Kobra, people started emailing me saying, “I wish there was a way to see the files I have collaborated on in the past." Only then did I program the dashboard.

Going "too minimum" is also a possibility. Yes, you can feel free to leave things out such as password reset (you can do that manually!), but don't make it so "bare bones" that no one uses it.

That's where the next part of MVP comes in.

The "Viable"

There is a reason that I saved "viable" for the end. It is by far the most complicated term in MVP. The problem is that you can't actually build a "Minimum Viable Product" on your first try.

Why? Because the definition of viable is: "capable of working successfully; feasible." How are you supposed to know if something is actually viable as a product until you have people using it?

The answer is: you can't.

So at this point in time, you are left "guestimating" on the exact features that you should launch with. Talking to prospective users is a good idea, but they don't always know what they want, and they may try to convince you to throw every single feature idea you have into the product.

To bring more focus to the term Viable, I want to dive into some other words that people often replace it with.

Desirable vs. Viable

Some people argue that you don't need a "Minimum Viable Product," instead they want you to build a "Minimum Desirable Product."

People that believe this are definitely on the right track, but we need not change the word, instead we need to understand the concept of Viable better.

A product that is "capable of working successfully" or "feasible" is a product that will gain some sort of market traction if released. By definition, that means that the product would be considered desirable.

So let's throw "desire" into the definition:

Viable: "capable of working successfully; feasible; desirable."

Valuable vs. Viable

Some people argue that MVPs "aren't good enough" because customers only care about Minimum Valuable Products.

Every product exists for a specific reason, and the reason that the product exists is the value of that product. There is value behind this Dr. Pepper I'm drinking because I find it refreshing, therefore I bought it.

A product that is not valuable will never make a sale, and even if it is free, it will never gain a user. The only reason anyone ever utilizes a product is because they find value in it.

Sounds pretty confusing, but it boils down to this: if you are out to build a product, it MUST provide value to someone, otherwise what on earth are you building a product for?

Sellable vs. Viable

There is yet another camp of people that replace the term Viable with Sellable. However, not every product is out to make a sale. For instance, let's say a Non-Profit organization wants to release a Software-as-a-Service product for free to help people volunteer.

In this case, the products goal is not even to be "sellable."

Now let's step back for a minute and think of the term Viable in Minimum Viable Product as bi-directional. Not only must the product be viable for the consumer, it must also be viable to you as a business.

That viability can come in a number of forms. Perhaps the Non-Profit just wants to increase their volunteer count, which would make it viable for them if they can indeed increase volunteering.

If you are running a for-profit business, a product is only viable to you if it can "keep you in the black." This means that it assists your business as a whole in turning a profit.

A product may be 100% Viable when looking at it from the consumer side at $5/month. But if you are losing $5/month on that customer, it is not actually a viable product from your side: the business side.

Tying it All Together

Let's go through each term in MVP one last time just to provide you with a sort of "checklist" to determine whether or not you are actually building a "Minimum Viable Product" and not the "Giant Potentially Perfect Product" you have in your head.

Minimum: You have narrowed down your feature list to only include the features that you think are necessary for your product to be considered viable.

Viable: Feature list is not the only thing that makes an MVP viable, you also have to take into account the bi-directional value to you as a business.

Product: Both the short-term ideas you have for your MVP and the long-term goals for this product as a business. This may include features that you are leaving out of your MVP.

Making sure that what you are working on is an MVP is the easiest way to make sure that you can build your product quickly and begin testing it as soon as possible.

I encourage you to spend some time on this step. One week about thinking about your product and turning your idea into an MVP could save you months of wasted time if your product doesn't succeed as well as you want it to.

Perhaps you have a complex product out there already, such as myself and the first version of Kobra. If that is the case, don't be afraid to scrap it and build an MVP.

The re-launch of Kobra in a more minimum state has produced around $3000 in sales, 900 users, 7000 files collaborated on and 20% of visits result in two or more people actually collaborating on a file.

There are a LOT less features in the current version than the old version of Kobra, but it turns out, sometimes less is more!

The Art of Not Wasting Time

The truth of the matter is this: us as developers, we have a problem. Every developer that I've ever met has it, I have it, and you probably do too.

When you're reading this, you should step back and try to look at yourself from an outsider's perspective. Is this something that you are doing? Because chances are, it is.

As developers, we hold our code and the things that we build to a high standard. A side effect of that high standard is that we like to "claim credit" for the code that we write.

When I was working at a medical analytics company, we encountered a severe problem. We were calculating so many statistics, and adding new behavior to the dashboard so frequently that we could no longer rely on a relational database structure.

So the other developer and I set out to solve this problem. We decided to switch over to a non-relational database, MongoDB, and store statistics in a format that would be easily expandable. However, as a result, the structure of the data no longer mimicked the dashboard.

I wound up building a relational ORM on top of MongoDB. Some people may think that sounds kind of cool, others may think that it was quite strange. But here's the reality: it worked amazingly.

Because it worked so well, I really like talking about it, and usually bring it up in interviews. Sometimes, it drastically backfires. The interviewer looks at me and asks, "you did what?"

The problem was, I was talking about the wrong thing. I was taking pride in myself and my code instead of talking about the one thing that actually mattered:

The end product.

Instead, I should have been talking about the complexity of the data and how we could add new analytics to our dashboard in as little as 20 lines of code, even if the data wasn't structured in Mongo "correctly."

I was valuing myself based on the code instead of the product I created.

Now, this time, I got lucky. I programmed something that really did have value, and that I couldn't have gotten anywhere else.

However, there exists a flip side. More often than not we actually spend time developing things that we don't really need to be working on, just because we can. Just because it's cool, and we want to tell our fellow developers how awesome the code is.

When I got my first "real job" as a web developer, I was spending a lot of time doing menial tasks:

- **Compiling my SASS into CSS**
- **Minifying my JavaScript files**
- **Concatenating my 30+ JavaScript files into a single one**
- **Restarting the web service when I made a change to my Python code**

I'm sure that I left out a few things from that list, but you get the point. Having to do 4+ things every time I made a single change was annoying.

So I did what every pride-conscious code-loving developer in my situation would do. I built a solution: devWatchr was born.

Whenever I got to work, I would boot up devWatchr and leave it running all day. It would watch for files to be saved, and then perform the necessary tasks. If I saved a JavaScript file, it would minify it, then re-concatenate all of the JavaScript files together.

I even built the system to be completely expandable. You defined your “watchers” in a JSON file, and used plugins like “SASS to CSS” or “Minify JavaScript” to perform the necessary actions. I threw it up on GitHub and called it “open-source.”

There was a brand new shiny problem, and I had fixed it! I was cool now!

If you’re reading this as a web developer, chances are you want to shoot me in the face right now.

I showed devWatchr to one of my developer friends, and he just looked at me and laughed. “Have you seen Grunt, man? You know it does all of these things already, and people have already written hundreds of plugins for it.”

“WHAT?!”

Sure enough, I did a quick Google search for “minifying Javascript files on the fly” and a result popped up showing me how to use Grunt to do it.

So here I was, left with a solution that I programmed, that I could claim credit for but no one would ever use it. Why? Because I didn’t do my research up front.

Did I get a little bit of pride out of it? Sure. It was actually a pretty cool system. However, I definitely didn’t feel good after finding out Grunt existed.

I felt sick to my stomach. I had spent an entire week perfecting this system, building it from scratch just how I liked it. It felt horrible.

Imposter Syndrome

This is something I think all developers do: we find a shiny problem, and we immediately think about how to fix it. And since we value ourselves directly related to the code that we write, we wind up building a solution from scratch.

We do this either because we didn't think to look if it already existed, or because we think "we can do it better."

We latch on to this problem so violently that it becomes the only thing we think about; we have to fix it. Because if we don't, who will?

This evaluation of our self worth based on our code, this is where imposter syndrome is born.

We think that other developers have programmed cool things, so if we can't program cooler things, we're worse than them.

That's just not the case here. The most important thing is that you have the ability to take code, even someone else's code, and turn it into something concrete.

It could be a mobile app, a web-app, or even a fancy command line tool. Your ideas and your execution are the important part, not programming the entire thing from scratch.

If one developer takes an entire week and builds a system to watch files for changes and then perform specific actions (me), and another developer Googles around a bit and finds Grunt, who is in the better position? Who is the "better developer" in that situation?

Sure, I programmed something. But the other guy was more resourceful with his time, and in turn, probably released his product before I released mine.

If you saved yourself some time by using someone else's code, I would argue that you are definitely a better developer than the guy who wastes his time programming a solution from scratch. No imposter syndrome there.

The 3-Step Development Process so you Don't Waste Time

When this finally dawned on me, my three-step development process was born.

The sole goal of this process is to make sure that you aren't wasting any time. That you are only spending your time developing the things that actually matter.

Here are the three steps:

1. **Find It**
2. **Buy It**
3. **Build It**

This is a simple system that you should use in every iteration of your project. For every feature you're thinking about programming you should look at the "Find It, Buy It, Build It" system.

"I need my program to do X, let me go through the Find It, Buy It, Build It process."

This can be used in tandem with things that you might have already read or learned about. If you are in to Lean Development or SCRUM or whatever new system exists by the time you read this book, you should still be using the Find It, Buy It, Build It process.

Lean contains an iterative process where you "Build, Measure and Learn." My three-step process should be performed inside of the "Build" step of Lean, or whatever the "I'm actually programming now" step of the framework you use is.

Now let's dive into the first step!

Step One:
“Find It”

The first part of this method is “Find It.” By “find it” I mean:

- **You’ve got an idea for a feature that you know needs to perform a certain task.**
- **That “task” is what actually needs to be programmed.**
- **Is there code out there that already exists that you can “find” to program a majority of the feature?**

This may seem incredibly simple at first glance, but it is the single most important step to this framework.

For me, I knew that Kobra.io needed to have built-in video and voice chat. Many developers already used solutions like Google Hangouts or Skype, but I wanted to make sure that people could chat directly in Kobra, thus simplifying the process of coding together.

However, I didn’t want to spend all that time reinventing the wheel.

So what did I do? I went out and I found a solution that I could use right out of the box.

I knew that something out there existed called WebRTC, which is a fairly new specification. I also knew that since it was so young that it was not fun to program, especially when it comes to cross-browser functionality.

I began Googling for “WebRTC plugins” and eventually landed on EasyRTC (please note that EasyRTC is no longer maintained). I actually tried out five different solutions, but finally decided to use EasyRTC just because I thought it was... well...the easiest to use.

So in this specific case, I managed to succeed in step one. I wanted video and voice chat, so I went out and “found” EasyRTC.

I was up and running in under two hours and with less than three hundred lines of code.

In my opinion, that was a pretty slick feature. Some developers that had early access to Kobra sent me an email once I pushed the update:

“Man, that’s really cool that you added built-in video chat! How did you do it?”

Instead of telling them “I coded this whole thing from scratch, it took me three weeks,” I put my pride aside and merely told them, “I found this library called EasyRTC and made it in two hours and under 300 lines of code.”

It’s all about being resourceful; it’s about using open-source code to your advantage.

It’s about coding as little as possible.

Googling is an Art

The truth of the matter is, there is a whole lot of open-source code out there, but you’re not always going to find exactly the solution you need. Sometimes it’s really difficult to find solutions to the specific problem that you’re encountering or the feature that you’re trying to program.

When I first showed my three-step process to another programmer in the industry he told me, “Matt, the ability to find these things: it’s a skill.”

So let’s go down the rabbit hole of Google, because sometimes you’re not going to find what you need right away. But you can’t give up quite yet; I want you to give it one more go.

One of the things that I wanted in the original version of Kobra was the ability to interact directly with a developer's server so that when they saved a file in Kobra, it would automatically be saved back down to their server.

I began Googling with "file transfers in Python" and found absolutely nothing useful. Either the code didn't do anything remotely close to what I needed it to, or the solution was way too complex.

But I definitely didn't want to program this solution from scratch, so I began to "boil down" the exact idea I was looking for.

Instead of using abstract concepts like "file transfers in Python" I started with something concrete. I knew that SFTP was what I used to transfer files back and forth between my local machine and my server; perhaps there was an SFTP solution? So I kept on Googling:

"SFTP Python Library"

There were definitely some libraries that existed, but they still all appeared too complex to me. Eventually I landed on a developer tool called "fabric."

This tool was built for deploying code. So if you work on a development server, once you were ready to push code to production, you would use "fabric" to do that.

It definitely wasn't the "perfect" solution. Kobra didn't need to have fully functional deployment scripts to transfer files to your server, but fabric was close enough that I thought I could figure it out.

Some developers out there may harp on me for "using a tool to do something it wasn't meant to do." But you know what? I got the feature programmed in about 800 lines of code and I couldn't be happier.

Remember, this is all about valuing your time. Don't be afraid to use open-source software, and definitely don't be afraid to use it for things that it might not be "advertised for."

An hour and a half of Googling is much better than twenty, thirty or forty hours of programming a low-level file transfer between two servers from scratch.

Your Network is Valuable

Google isn't the only resource that you have available to you. There are armies of programmers out there that have already been slaving away at keyboards, and chances are at least one of them has encountered what you have.

Here in Milwaukee, I spend most of my day in the #devmke IRC chat room. There are about 30 developers on at any given point in time, and I've met most of them in person.

Chances are, if I have a problem, one of them knows the solution.

Focus on building your network, go to developer events for technologies that you think you may need to use in the future.

Remember how I said developers really pride themselves in the code that they have written? You can "use that to your advantage."

Ask developers what they have been working on, and I'm sure they'll tell you. These connections are priceless.

Even if you're not much of an "outgoing programmer" there are plenty of communities online that you can find. StackOverflow comes to mind, but you should search for IRC chat rooms centered around your physical area or forums that you can build repertoire in.

Didn't find a solution?
Try Again.

Go back and try again, yes, I'm dead serious. An extra hour of searching for a solution or asking other developers could save you dozens of hours down the line.

But if all else fails, you have my permission to move on to step two now.

Step Two:
“Buy It”

So you didn't find any open-source software that can help you to program your next feature. Now you have arrived at the step where I'm going to tell you to spend some money.

But before I do, I want to talk about you a little bit. Yes, you.

Your time is worth money.

That is probably one of the single most important sentences in this book, so I'm going to say it again:

Your time is worth money.

"How much money?" Why, I'm glad you asked. Let's just say that you make $20 an hour. You may make more, you may make less, but that's beside the point.

You make $20 an hour, and a feature is going to take you 20 hours to program. I'll do the math for you. That means that the cost of that feature is $400. If the feature is going to take you 40 hours to program, you are now at $800.

This is grade-school multiplication; it's not rocket science. However, not everyone is going to calculate their hourly rate in the same way.

If you are working a 9 to 5 job like I am, chances are you value your time at the hourly rate that you make at your day job. If you're a consultant, you might value your time at a higher rate. No matter what way you calculate your hourly rate you need to be sure to have one.

Now I don't know about you, but I value my time much more than my money. If my goal is to launch a product in a weekend, I'm extremely concerned about every minute.

So let's say that programming the feature you're working on is going to cost you (your development time * your hourly rate) = $400. If I can find some closed-source code or a service that only costs me $50, you better bet that I'm going to buy it in a heartbeat.

The reality is, since your time is worth money, you have effectively saved $350. Now, I know what some of you are thinking right now, "Matt…I spent $50, I didn't save anything," which brings me to my next point:

Other developers' time is also worth money.

There are two hard-truths that I want you to break yourself of:

1. **If a closed-source solution or service exists that does one specific thing, chances are, you cannot do it better. Especially not in a short amount of time.**
2. **Not everything comes for free. Your time is worth money, but other developer's time is also worth money. You're not going to find open-source for everything.**

We're talking about building software products in a single weekend, about getting them out to the public as fast as possible.

You are not going to be able to code a perfect, or even a complete, solution to every feature or problem you encounter.

If someone else has already done the hard work for you, and they want $50 for it, give them the $50 and move on.

Literally the most important feature in my side-project, Kobra, is the ability to code collaboratively. Without that, it wouldn't even exist.

That is an extremely complex feature to program. I would have to go out and learn object-transforms in order to do it, and my database would have to be set up in a way that would actually allow me to send data in real time.

Long story short, it would be a complete crap-storm if I had to program it myself. So I began to Google around for "real time code collaboration."

Eventually I found Firepad, which is a real time collaborative text editor by Firebase. Turns out that it can function perfectly with the open-source online code editor ACE.

Technically Firepad is an open-source project, but Firebase is a paid service with the first tier starting at $50 a month. However, Firebase's service provides me with a real time database too.

Together, ACE, Firepad and Firebase provided me everything I needed to incorporate a real time editor into Kobra. Got a guess how many lines of code I had to write to get the editor functioning?

Ten.

I didn't have to worry about object-transforms, I didn't have to worry about hosting my database, I didn't have to worry about real time speed, I didn't even have to worry about hosting my static content. Effectively Firebase replaced my entire server.

Yes, it costs me $50 a month to keep it up and running, but I can't even begin to imagine how many hours I save by not having to deal with any of that.

Defeating your Manager on Purchasing Decisions

One of the things I haven't mentioned while talking about the three-step process is the fact that you can use it for ANY development project, even one at your day job.

However, the “Buy It” step is the hardest one to execute in an environment where you have a manager or boss that has to approve all purchasing decisions.

But there is one simple trick that you can use to get your manager to buy pretty much anything you want, even if you’re in the middle of a hiring freeze or your company is lowering your budget.

You know that simple equation that we learned?

Your Development Time * Hours It Will Take To Complete = Total Cost of Feature

This is where it really comes in handy. If you just show hard numbers to your manager and explain them to her, chances are they will buy you whatever you need.

At my current job and my previous job I succeeded in getting an $800 purchase order approved in a matter of seconds. Both jobs required the use of extremely beautiful and interactive graphs in-browser.

Frankly, I don’t want to learn how to program SVG or D3 from scratch, so I found HighCharts.

HighCharts is free for open source, but commercial use is $400 or $800 depending on the version you get. Here’s how the conversation with my boss went:

I poked my head into his office, I was pretty nervous, and asked him if he had a second.

“Hey Bob, it’s going to take me a few weeks at least to learn how to make these graphs you want, but I found a pretty simple library we can use for $400.”

“Okay, here’s my card, bring me back the receipt.”

You've got to be kidding me, I was nervous over that? At my current job the conversation was even better:

"Hey Sue, I have three years of experience with HighCharts, it's going to cost $800 for us to buy the library."

"Sweet, bring me back the receipt, I'll explain it to the CEO."

In both of these situations, all I did was mention the amount of time that it was going to take me to do something, and offer a monetary alternative.

For larger purchases, or bigger projects, you may have to actually write out the formula and plan to show how much money it would cost you to do from scratch and how much it would cost to buy it. But in the end, less hard work for you!

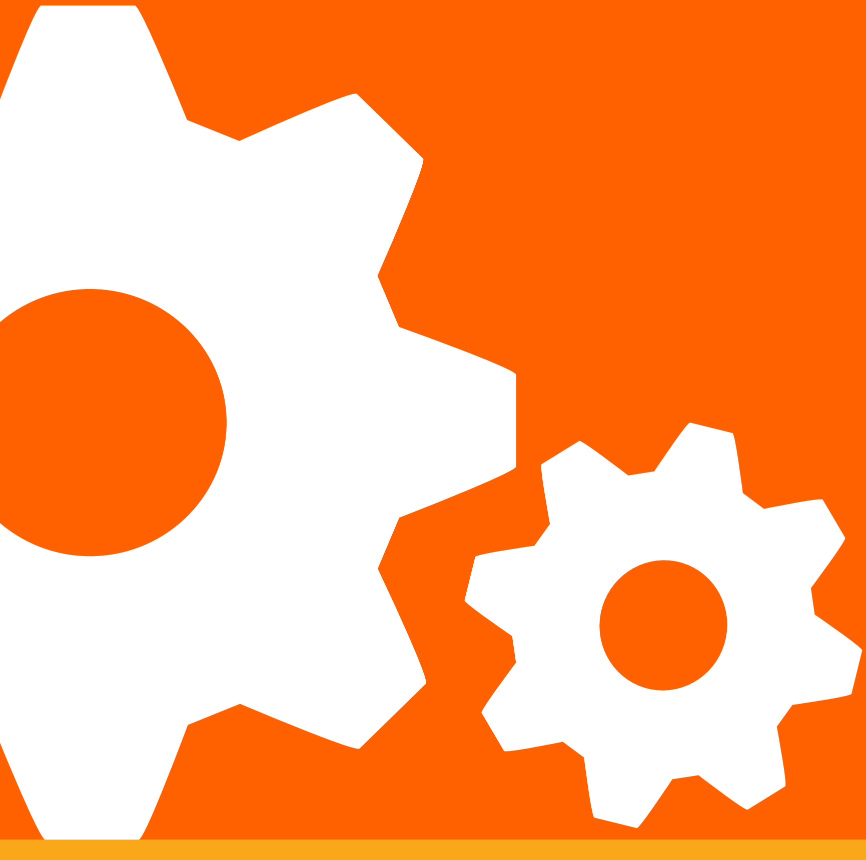
"I Still Can't Find a Solution"

At this point, you better be damn sure that one of three things is true:

1. **The piece of code you're working on is actually business-logic or application specific.**
2. **You've spent at least half a day, if not more, searching for a solution and asking your network if they know of anything.**
3. **You HONESTLY think that the feature is going to take you $400 worth of hours to build, and the solution you found costs $10,000. Honestly is the key word here.**

Got it? Good.

Step Three: “Build It”



If you've completed steps one and two successfully, the only thing left to "build" is the actual application and business logic for your mobile app, web app or other tool.

Essentially, what you are doing at this point is building the "spider-web" that connects all of the open-sourced software from step one and closed-source or services from step two into your actual product.

Here's a list of everything I found before I even got to step three for Kobra:

- **AngularFire:** The JavaScript framework that consists of AngularJS backed by the Database-as-a-Service Firebase.
- **Firepad.js:** An open-source project by Firebase that has collaborative editing built into AngularFire.
- **Firebase SimpleLogin:** A login service provided by Firebase that automatically integrates with: Email/ Pass, Facebook, Twitter, Google and GitHub. I hate programming authorization, so this was an awesome find.
- **ACE Editor:** A beautiful open-source code editor that comes built in with syntax highlighting support for 100+ languages, themes, code folding, keybindings, etc.
- **EasyRTC:** An easy library to use so you don't have to code WebRTC Video/Voice chat. Like mentioned earlier, this library is no longer maintained.
- **Stripe:** A really easy payment system that was built for programmers.
- **Bootstrap:** Twitters design framework that has great looking buttons, modal dialogs, etc.
- **UI Bootstrap:** AngularJS directives to replace the jQuery from Bootstrap.

As you can see, pretty much 95% of the features and code I needed to write was already done for me. All I needed to do was tie all of the pieces together, make it look how I wanted, add a few minor features, and launch!

So I programmed the "stuff in the middle" of all the pieces I found, added a few things, and **thirty-six hours later** I had myself a Minimum Viable Product.

Here are some tips that I've found very helpful for the build-phase:

Know your Framework

Frameworks exist now for almost every single language. If you are still programming in raw JavaScript/jQuery for example, you may want to try checking out AngularJS, Backbone+Marionette, Ember, etc. If you are using Python I suggest using Django for larger projects or Flask for smaller ones.

If you're not programming web or server-side, there are still plenty of frameworks available in other languages you may be used to such as C, Java, etc.

The point I'm trying to make here, is that you should definitely be using a framework; no matter what language you are programming in.

They exist to make your life easier, providing structure around your app and helper functions to do many menial tasks.

However, I also don't recommend jumping into a time-sensitive or weekend project without first knowing your framework well.

I suggest that you spend some time with it: develop some very small programs to get used to it, follow online tutorials or buy a book on the framework.

Jumping into a project without already knowing the framework can be a nightmare. Perhaps you begin coding, but 2000 lines later you realize that it would have been much easier had you organized your code differently.

When building products, know your framework like the back of your hand. When building small apps for fun, pick a new framework to learn.

Know your Tools

Another point to mention is that you should also be very comfortable with your tools. This includes your actual code editor, but it may also include things such as:

- **Your server stack**
- **Your deployment & version control tools**
- **Your graphics programs**
- **Your actual development process (live debugging, etc)**

You don’t want to be caught with your pants down if you’re working on an Android app during a hack-a-thon and you’ve never actually compiled one before or you’ve never tested live on a phone.

Just like your framework, you should have knowledge of all of the tools you use before getting started on an actual product.

Again: build products when you have knowledge, build fun stuff when you’re learning.

Don’t Switch Frameworks or Tools because of “Shiny”

Just because a new framework or tool looks awesome, doesn’t mean that it doesn’t come with a cost. That cost is the time that it takes you to learn the new tool.

If you are already extremely comfortable with a language, framework or tool chances are you can develop in it extremely quickly even if it may not be “the right” framework or tool for the job.

Remember, it’s not about finding out which perfect set of tools to use; it’s about finding THE tools YOU can use effectively.

You may find your “golden combination” of tools that you like, and you’ll get comfortable with them. Right now mine happens to be AngularJS and Firebase with a small Node.js server on the back if I need it.

Once you harness the power of that one killer combination that works well for you, I guarantee you that you are going to be able to launch MVPs in record time.

Build Pieces to be Re-Used

Let’s say you didn’t find any open-source or closed-source software (go back and check again!), I highly recommend that you build pieces to be re-used.

You don’t want to write a large chunk of code for use in only one project. See if there is a way to write the code as a plugin or extension so that it can be used freely in other products you create.

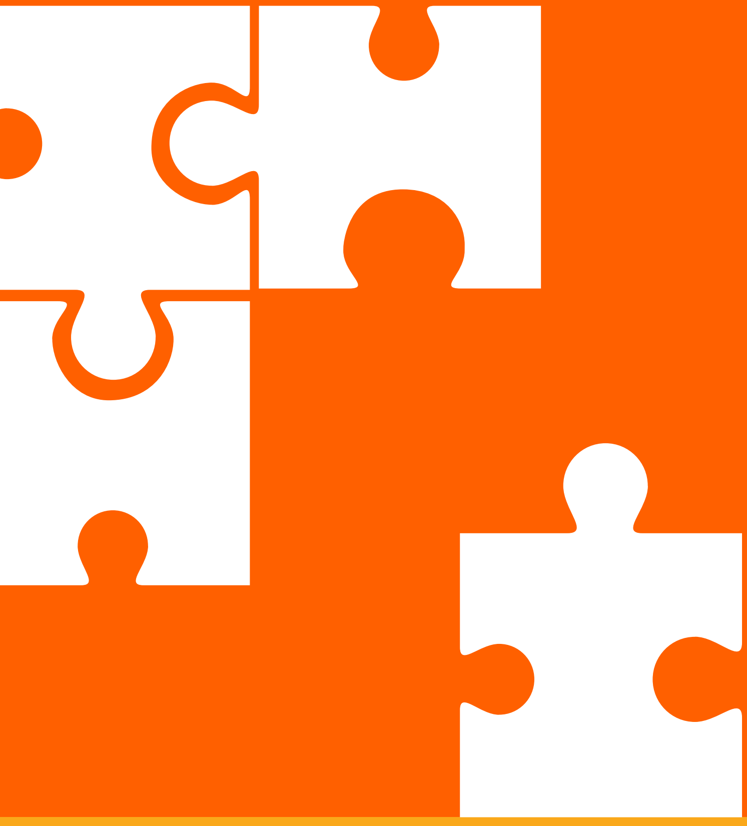
There is an excellent article called “Sell your By-Products” by Jason Fried of Basecamp. In the article he talks about how it is extremely beneficial to actually sell these pieces that you create from scratch.

Whether or not you actually put a price tag on your code doesn’t matter. But if you force yourself to treat the custom code like you were going to be selling it, this will also force you to design it to be used across multiple projects.

For instance, if you “need” a custom icon set for your product perhaps you could sell that icon set on an online marketplace?

When I wrote the MongoDB Relational ORM I talked about in the “not wasting time” section, I wrote it to be completely separable from the project. If I encounter another project where I need to calculate a large amount of statistics and show them in a dashboard, chances are I will break out that ORM and use it.

Putting It All Together



So you have learned the benefits of coding as little as possible and not wasting time, then we talked about my three-step development process.

However, just utilizing these methods isn't going to allow you to build your app in a single weekend. I put a lot of planning and work in before the weekend starts so that I know I'm not going to be wasting any time floundering around, twiddling my thumbs or not knowing what to program next.

I want to show you exactly how I tackle a new project by utilizing all of the methods in this book. Think of this chapter as your "homework" for before you start working on your next product.

Phase 1: Research and plan out your MVP; make sure it is actually an MVP!

During this phase, you should write down what your product's purpose is. What pain point is it going to solve? Why is it important that you build this specific product?

Then you will research your competition. Make sure that you are different enough. You should have a USP or "Unique Selling Proposition." This is what would make someone use your product over the competition.

It can be as simple as "my product will be easier to use" or as complex as "I found out a completely different way of accomplishing the same task that is more efficient."

Be sure that you are jotting down everything that you find out in a notebook or word document. It's important that you have as much solid information as possible. The more you know before you start programming the better.

The next thing you must do in phase one is make sure that you are indeed making a Minimum Viable Product. Remember the definitions of “Minimum,” “Viable,” and “Product:”

Minimum: You have narrowed down your feature list to only include the features that you think are necessary for your product to be considered viable.

Viable: Feature list is not the only thing that makes an MVP viable, you also have to take into account the bi-directional value to you as a business.

Product: Both the short-term ideas you have for your MVP and the long-term goals for this product as a business. This may include features that you are leaving out of your MVP.

Phase 2: Cycle through “Find It” and “Buy It” for each feature on your MVP list, and then try each solution!

By this time, you should have a list of features that you are going to include in your MVP. For each of those features, I want you to go through only the “Find It” and “Buy It” steps.

You should find the open-source or closed-source software/services that are necessary to build each feature.

For each possible solution that you find, I want you to do two things. First, be sure to write the solution name next to the feature on your list. It is possible that you may have many options for a single feature, such as when I was looking for a

WebRTC Video and Voice chat solution.

The second part is the most important. I want you to actually program something small with each of the technologies. The time spent playing with the technologies is not part of your weekend development time.

This is time that you should spend learning each technology. Not only will this allow you to pick the best or easiest solution, but also it will make you familiar with the technology so when you actually sit down to program your product you don't have to be learning at the same time.

When I was trying out WebRTC solutions, I programmed a simple HTML page where all I did was import each library, kick off the WebRTC with a hard-coded "room name" and give the URL to a friend. I made sure that we could indeed see and hear each other.

Then I made sure that I could exit the chat and join again without reloading the page. Turns out that this is actually pretty difficult in WebRTC, so I immediately threw out two of the possible solutions because I couldn't figure out how to perform that task.

Even if you are competing in a Startup Weekend or hack-a-thon event, I still recommend you play with the technologies you plan on using before the event. Like I said, the most important part here is that you are not learning while you are trying to develop quickly. Instead you are learning during your own free time on "mini projects that don't actually matter."

Once you have narrowed down your solutions to only one-per-feature, it is time to move on to the next phase.

Phase 3:
Actually "Build It."

Phase 3 is your "Building Software Products in a Weekend" phase. By the time you reach phase 3 you should already know each piece of code you have to write in your head.

Since you have already tried all of the technologies and open/closed code that you are using, you know the intricacies of each one, and you know how the "puzzle pieces fit together."

The better you know all of the frameworks and technologies that you're going to use, the faster you are going to be able to develop your MVP. That's why phase 2 is so important, and more often than not, developers tend to skip phase 2 and just dive right in.

That is a sure-fire way to shoot yourself in the foot before you've even started.

At first, you may not be able to finish up in a weekend. But as you get more and more comfortable with this process, you'll be developing products faster than ever before.

Phase 4:
Rinse, Wash and Repeat

Alas, even after you have built your MVP, the fight is not over! After you get your first customers or users it is time to start interacting with them. I usually send out surveys to my users to ask them exactly how they are using the tool.

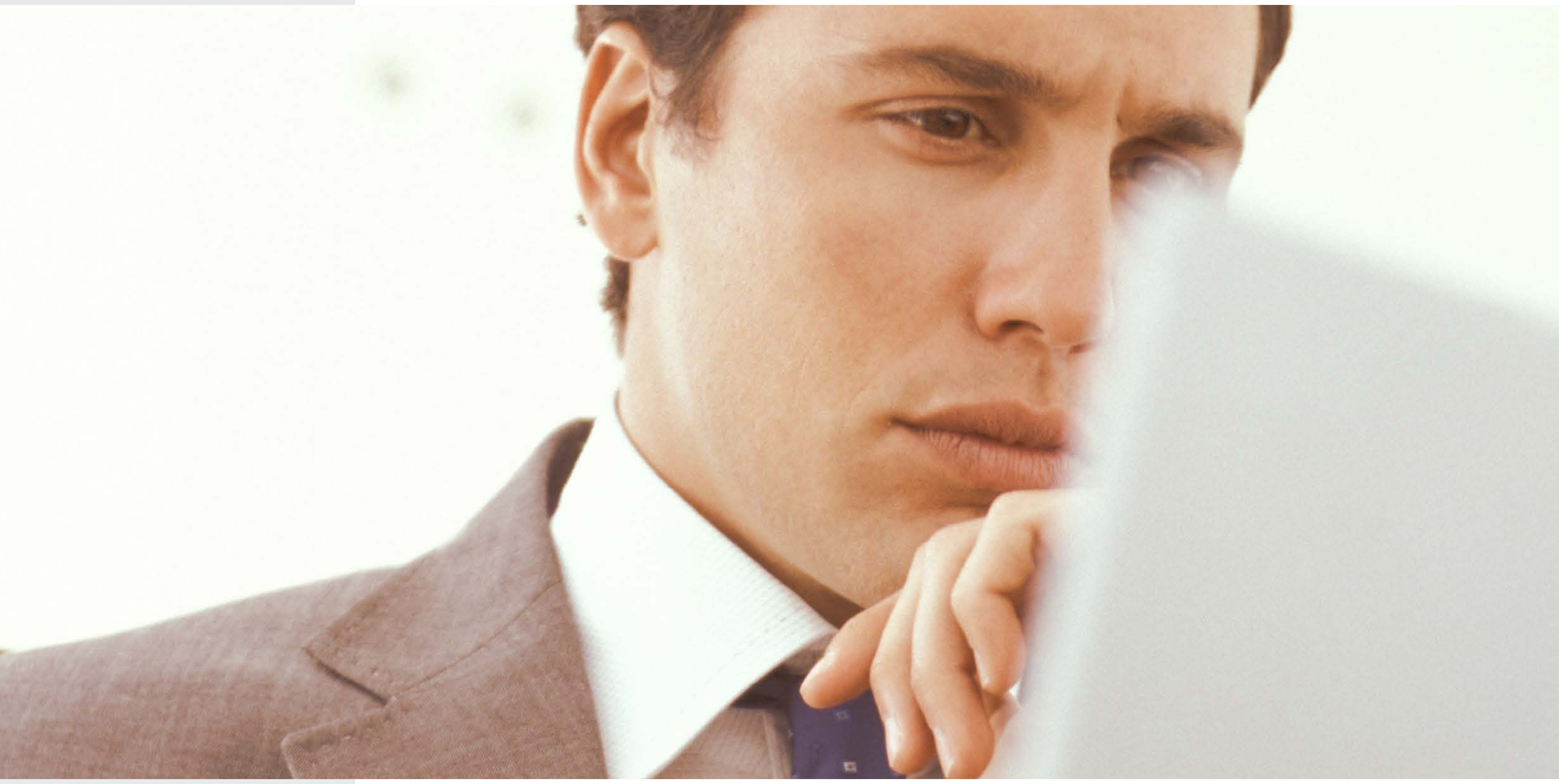
For Kobra, I asked them "Are you working on large projects, or small projects?" to determine whether or not they may get benefit out of me adding in the ability to work on multiple files at once.

If you have “power-users” or users that use your product more frequently than anyone else, establish a personal relationship with them. Hop on Skype or Google Hangouts and talk to them about how they use the tool.

The more you interact with your users, the better your product will become. Instead of starting out with your “Giant Perfect Product” you are building one as you go. Doing this will make sure that you are building exactly the right features at exactly the right time.

Going about developing a product this way will not only wind up producing a better product, but it will keep your actual development time minimal in the future as well.

Instead of focusing on the entire product all at once, you are interacting with users and adding one feature here and there when you think it is important or it gets requested often.



Where to go next

Remember, a product is never actually "perfect." It can always be improved! Just because the weekend is over doesn't mean your job is done.

There is always more to learn.

I'm writing new blog posts and recording podcast episodes on building, launching and marketing products all of the time.

If you're not already signed up for my email newsletter, please check it out here:

https://MattKremer.com/newsletter/

You can also subscribe to my podcast using iTunes, Stitcher or your favorite podcast app:

https://MattKremer.com/podcast/

I'll be sure to let you know when I release new content. Here's one particular article that you may find valuable for actually launching your product. It goes over three different strategies to use to get more pre-sign ups before you even launch!

"3 Launch Strategies to Double your Pre-Signups"

https://mattkremer.com/3-launch-strategies-to-double-your-pre-signups/

And remember, I'm always here to help. Feel free to shoot me an email at **matt@mattkremer.com** with any questions you have about this book, or about building products.

Did you enjoy this book?
Share it with your friends!

Thank you for taking the time to read “Building Software Products in a Weekend!” I hope that you enjoyed reading it as much as I enjoyed writing it for you.

It would mean a great deal to me if you shared this book with your friends and fellow developers. Feel free to forward this PDF to them or share it on social media using the link below:

https://MattKremer.com/sharebook/