Adam Darrah

Dropkiq Your Liquid

Supercharge Your B2B Engineering Team Using Liquid

dropkiq

Table of Contents

Introduction 3
- Counter Strike Source 4
- Starting Ruby on Rails 5
- My Software Career 6

Chapter 1 8
- **What is Liquid?** 8
  - Liquid is a Programming Language 9
  - Liquid Errors Gracefully 10
  - Liquid is Portable 10

Chapter 2 11
- **How I've used Liquid in my career** 11

Chapter 3 15
- **Why Use Liquid?** 15
  - Separation of Concerns 17

Chapter 4 19
- **Building Blocks** 19

Chapter 4.1 21
- **The Custom Field System** 21
  - What Does a Custom Field System Look Like? 22
  - Custom Field Values 24
  - Validation 25
  - When to Use a Custom Field Vs. a Database Column 26
  - Custom Field Final Thoughts 27

Chapter 4.2 28
- **The Conditions System** 28
  - Example Usage 30
  - What About No Conditions? 33
  - Short Circuiting 34
  - Condition Weights 34

Chapter 4.3 36
- **The Rule System** 36
  - The Rule System Final Thoughts 40

Chapter 4.4 41

Email Template System **41**
Example Email Template System 43
A “War” Story 45
Beyond Email Templates 45
Chapter 5 **46**
Using the Building Blocks **46**
Conclusion **51**
Impact Vs Effort 52
Get the Highest Impact From Your Team 53

Introduction



I'm writing this book to share some ideas about Liquid, which has been a tool that I've personally relied on my entire software career. It's interesting to look at how companies are solving problems today. After you've been doing software for so long, you realize that most problems are not unique. Most companies have the same challenges. Interestingly, some of the solutions that I've come to rely on remain difficult problems for many companies.

In this book, I will share with you some of those solutions so that you may use them yourself. I will not be providing any source code, only the ideas. The ideas presented here can be implemented in any language for any business domain.

I've been extremely fortunate in my career to have worked with some incredibly talented people. Even though I've personally put hours of blood, sweat, and tears into these solutions myself, I could never claim that they were my ideas. Many of these ideas were implemented on teams that I've worked with by many different people.

With that said, however, many of these ideas are not unique. As I look around at many tech companies, most of them are using many of these same solutions in their products. It's validating to see that since that would mean that the solutions are working! I'm writing this book for people who are like myself when I started my software career. This book is for people who may not have the advantage to be surrounded by the great leaders that I had, and it's up to them to figure it out on their own.

First things first, I should start by talking about my own personal history...

Counter Strike Source

When I was in high school, I played Counter Strike Source every waking moment possible. I didn't know it at the time, but this was my first introduction to leadership and people management.

I recruited a team of players, and we competed in a league called CAL. The only thing I ever thought about during this time was my counter strike team. There was one thing that was unique about my team that made us different from everyone else. Other teams had a very cut throat mentality. It was all business and winning oriented. Although we were competitive, our team was focused on growing together. It was a slow process, but after about five years the team I had started competed as a professional esports team for one season.

It was during this time that I got interested in computer programming. The team needed sponsors, and you could only get a sponsor if you had a website. Thus, I wanted to create a website for my "clan". This requirement set a series of events in motion that resulted in going to college for computer science and ultimately a career in web development.

Starting Ruby on Rails

During college, as I was taking my computer science courses, I was far more interested in entrepreneurship. I was inspired by a story I had read about the development of Twitter. Two developers building an app in two weeks and seeing explosive growth. I was so inspired! I thought that I should do the same thing, so I started learning Ruby on Rails.

My goal was to build a music service. This was in the early days of Grooveshark where music wasn't nearly as accessible as it is today. The goal of my application was to allow the user to enter any artist, and then the application would instantly give you a playlist (powered by Grooveshark) where you could listen to their top hits. I spent months crawling music sources, building a database of artists and their music so my service could give results instantly.

I thought I was going to change the music industry... Sadly, that never happened.

At that time, Rails was still in its infancy. Not a lot of people were working with it in Indianapolis at all. In one of my computer science classes, I saw a fellow student

working on a Rails project during class. One thing led to another, which resulted in an internship and a job at a Rails shop that I would be at for nearly ten years.

My Software Career

Reflecting on where I was when I started my entrepreneurial journey... I had no chance. I knew so little at the beginning, and I've learned so much in the past ten years. Over that time I've worked on many failed side projects and startups. I've also been a part of a few successes, and have been a part of a few acquisitions.

Here are some of my notable achievements from my career so far:

- A friend recruited me to work on his startup that had tens of thousands of users. In my time, I helped with many enhancements, but the one that I am most proud of is the premium user subscription service that we started. By the time I left the company, our subscription service was generating thousands per year in revenue and quickly growing.

- Together, we built an "ELO ranking engine". This was a one-of-a-kind system that is capable of ranking players for any game using the ELO ranking system. What makes it special is that it is the only implementation of ELO that is capable of editing or deleting the match history. This is a huge programming problem because any modification results in a chain reaction and impacts every match and player after the edit.

- I was involved in a nonprofit that raised millions of dollars. In fact, I was completely in charge of the technology and volunteer engineering team for a few years. Although I'm proud of what we accomplished, this role took a toll on my health because of how demanding the responsibility was. I learned a ton from this experience though.

- I built a plugin for Minecraft that allowed the user to manage minecraft builds (schematics) in a similar way to Github. From the game, you could instantly

upload or download schematics to or from the world to the website. From a programming perspective this was some of the best work that I've done, but the project never gained any traction.

- I learned how to manage people from one of the best software managers in the business. At first, I didn't want to be a manager. However, reflecting on the experience I learned so much about how to work with people.

- During my time as a Software Team Lead, we created one of the most challenging technical projects of my career. It was a BPMN workflow system capable of allowing users to essentially program through the user interface.

Reflecting on everything I've done, I'm so lucky to have met the people that I did and had the opportunities that I had. At the time of writing, I have left all of those people behind to start my own company, which is one of the hardest things I've ever done, but also a dream of mine since I was a teenager. It is only with the lessons that I learned from those people that I am ready.

Even though I wasn't a part of a huge company that is a household name (...yet), I have worked on many engineering teams with a wide range of people, problems, and industries. I've dedicated the last ten years of my life to truly understanding what it takes to be successful in the world of software (in particular B2B).

One thing that I have come to realize about my career is that there are a handful of tools that I rely on for everything that I do, and one of those tools is Liquid.

This book has many high level ideas about the possibilities of Liquid and how it can supercharge your development team and company. There will be little to no code. It will be most beneficial for software engineering leaders who are solving problems for their web portal B2B Saas companies. If you feel like you're working hard, but never getting ahead, then this book may be exactly what you need.

Chapter 1

What is Liquid?



Liquid was originally created in 2006 by Tobias Lütke, the Founder and Creator of Shopify. The original purpose of Liquid was to create a safe way to expose data so that designers could independently build Shopify themes. Lots of companies today use Liquid in many different ways. If you keep an eye out, every once in awhile you'll see some curly braces ({{ }}) and spot its syntax in the wild.

From a programmer's perspective, Liquid is incredibly useful. Businesses (including Shopify) want their software engineers to be focused on developing the next big feature to increase the capabilities of their products. They don't want their engineers building out custom themes for every new customer that is being onboarded. Software engineers are expensive and Shopify knows this.

Liquid solves that problem. With Liquid (and the right toolset), any person with the right knowledge can build out a theme that is customized for the customer. The only time the programmer needs to be involved is if there is an enhancement to the tools themselves that is needed or there is a bug that is preventing the theme from working correctly.

Liquid also solves a problem for programmers around security. Let's say that Shopify orders contain the user's credit card number (which they likely don't for PCI compliance rules). Programmers need to be sure that designers can't access this data since that is sensitive data that shouldn't be shared. Liquid gives programmers the ability to whitelist what data should be exposed through Liquid (and anything not whitelisted is kept a secret).

Liquid is a Programming Language

It says on the official Ruby Github page that Liquid is a "template engine". However, after digging into the code, I've realized that Liquid is a legitimate programming language. In fact, at a previous role we used to do "Dogfooding Challenges" where we would attempt to use our technology to solve programming challenges. Sometimes during these challenges, engineers would use Liquid to solve some pretty complex problems.

If you dig into how Liquid works, you will see that the raw Liquid code is tokenized and parsed, just as a compiled programming language would be. In fact, if you're interested in learning about the inner-workings of Liquid itself, I recommend starting by learning about compilers. This is a pretty good writeup to start with.

Liquid supports variables, conditional statements, and looping in a similar way to any programming language.

Liquid Errors Gracefully

One of the most critical features of Liquid is how errors are handled. In a normal programming language, when something goes wrong the whole program may crash. Liquid was designed for designers and users, who may or may not know what they're doing. For Liquid, when something goes wrong, the error is caught and the result is that nothing is output.

As with all design decisions, this architecture has some pros and cons. The biggest advantage is that small mistakes don't crash the whole program. Instead, Liquid will continue on as if nothing ever happened, and *hopefully* your users don't notice. The biggest disadvantage is that it is not always obvious why your liquid isn't working. Other programming languages will often attempt to give you a clue about what is wrong using error messages.

Liquid is Portable

Liquid was originally built in the Ruby programming language and was packaged as a "Ruby Gem". However, since 2006, Liquid has been ported to many languages. Chances are, if you're working in a major programming language, there is a way for you to use Liquid in your application. At the time of writing, there are 16 ports listed on the official "Ports of Liquid to other environments" page.

Chapter 2

How I've used Liquid in my career



We had a long standing rhetorical question at my previous job. A question that has come up many times during my career: Should our software set the standard for how people should work, or should our software adapt to how people already work?

In a previous role, when the company first started, the team saw an industry with many different ways of working and felt that we could standardize things. We attempted to introduce conventions for working best, just as Ruby on Rails had done in the programming world. Convention over configuration. This worked for some customers...

However, when you're a startup, you have to do what it takes to survive! I've seen this many times where the team has a vision of what they want a product to be, but then the market has different ideas or needs. It can be hard for software people to accept, but you have to adapt to make the business work sometimes.

As time went on and we were scaling our business, we saw that that was not going to work. We were dealing with big companies that expected us to accommodate their needs and their ways of doing business. From a business perspective, we couldn't afford to pass up on opportunities because they weren't ideal customers.

Over time, we developed several key "tools" that made it possible to adapt our software to our customers:

1. Every application that is handling a wide variety of customers will at some point need a 'Custom Field' system. This is especially true if you're working in the enterprise with a wide variety of customers. A custom field system allows you to add additional fields to store your customer's data without continually adding more database columns. This allows each customer to have fields specific to their needs. As a contrived example, if your customer is a brick and mortar store, they might have a field called "store_location" to track where each employee works. Another customer that has remote employees would not require this field since that wouldn't make sense for them.

2. A conditions system can turn functionality on or off depending on logical statements that are configured through the UI. Many companies have this functionality, such as Zendesk. This is a fundamental and necessary feature to build in the flexibility that is needed when business logic gets hairy. Business logic can live in data (through this condition system), rather than hard coded in the codebase. This system can be powered by Liquid so that it is possible to compare any data that is wanted.

3. A rule system can handle special requests from customers around orchestration. It is a "pub sub" system that can be triggered at certain times from the application, and then take certain actions. Let's say that a customer wants their new users to be notified via email when they sign up. Written in the codebase, when a new user signs up, it will broadcast that event through the system. Then, any Rule that is listening for that trigger knows that the event happened and will take action. In this case, the action can be sending an email.

4. Finally, an Email Template builder allows your application to send highly customized emails depending on your customer's preferences. This allows you to brand their emails or insert specific content depending on the customer's needs. One thing I've learned over the last ten years of my career is that email is serious business. Users get upset when they receive too many emails, and customers get upset if emails contain incorrect information. This system allows you to be flexible to your customer's needs.

This set of tools provides the building blocks you can use to implement any customer within your application.

One of the last projects that I worked on at my last job was to build a BPMN Workflow system. This was a domain-specific workflow system that was built to solve the needs of the business. However, in reality, the Workflow system we created fundamentally was not that different from systems such as Zapier. The goal of the workflow system was to create an environment in which the customer could take control of the technology themselves. This was mutually beneficial for both the customer and us. For

us, the cost of customer support for customer specific changes would go way down with this technology. For the customer, they would get the results from the software themselves way cheaper and quicker.

This kind of system can be a long term goal for your organization. However, building out this kind of a system is a huge investment for both time and money. This may be overkill in the beginning stages of your company. In the beginning, you just need something that gets the job done as cheaply and efficiently as possible.

Chapter 3

Why Use Liquid?



I can remember that one of the first customers I ever worked with had an office based in Michigan, even though it was an international company with offices all over the world. Our company would ship items to their employees all the time, but for some reason anything shipped to this office needed to send a different email.

This was the early days of the company, and the first time we were trying to accommodate such an unusual request. Since it was the early days, we would do anything for their business and failure was not an option. Without knowing any better, we simply added an "is_in_michigan" boolean database column to our Order model.

As you can imagine, logic began to spread through our application. Everywhere, you would see "if order.is_in_michigan?". I remember, a mentor of mine used to always say, "Whenever you make a decision, think about what would happen if you make that decision 1,000 times.". What they meant by that was, imagine that we got 1,000 customers that all had a special rule and we solved that problem for all 1,000 of them by adding another boolean. As you can imagine, that would quickly become an unmaintainable nightmare!

When your application deals with many different customers, separating the data and the business logic is extremely important to scaling. With the tools mentioned here, Custom Fields, Conditions, Rules and Email Templates, it becomes possible, and even easy, to solve these problems. Liquid becomes the glue that holds all of these pieces together and allows each subsystem to communicate.

Separation of Concerns

Your business should work like Shopify.

Shopify understood that it was not scalable for their engineering team to create a new theme every time they got a new customer that wanted one. Their software engineers would be working themselves to death to build out new themes, which would not allow them to be developing new features and pushing their business forward.

Today, there are hundreds of Shopify Experts and companies that know how to use their theme builder tools. If you're a new company to Shopify and want a custom theme, that theme can be built by someone and Shopify never even has to know about it!

Valve software is another company that understands this concept. Valve has a team of designers who make hats for their video game Team Fortress 2. They have also released software so that freelance designers can make hats and sell them to people in the game. I once heard Gabe Newell, Co-Founder of Valve, say that their company works hard... but they will never be more productive than their community of content creators.

Even though you may not have a large community of freelance designers or developers, your company can still follow this lead. Your company should have two teams that work together: An engineering team and a configuration team.

The engineering team should be focused on creating the building blocks of your application. The engineers are responsible when a customer requests changes that are not possible with the current toolset. The engineering team is also responsible when things go wrong and there are bugs. You don't want to have your engineering team building customer specific requests when possible.

The configuration team works using the tools within the application that the software team has built. They configure the application to provide the experience the customer

wants. They can gather requirements from customers and build completely independently from the engineers.

Why is this important for your business? One word: money. Software engineers are expensive. Doing this configuration work still takes skill and training, but the learning curve is way steeper to become a software engineer than to learn how to use these configuration tools. The ultimate dream of companies is to achieve what Shopify has. That is, when your users become experts themselves and you don't need to pay people at all to configure your software.

Lots of companies in the world rely on this kind of separation of responsibility. With this team-architecture, Liquid is the glue that holds everything together and provides a common language. Liquid provides access to the data in a safe and flexible way so that the development team and configuration team can work together seamlessly.

Chapter 4

Building Blocks



In this next section, I will be going over each of the four building blocks in detail.

1. The Custom Field System
2. The Condition System
3. The Rules System
4. The Email Template System

We'll go over what each system is, the benefits, and how you can take advantage of this system in your application. Our goal is to take your team from being "busy" to being "productive". We'll do this by building the right tools for the long-term. If you feel like your team has a huge influx of work every time a new customer comes on board, then these systems are perfect for you!

To reiterate, I will not provide source code for these systems. However, I will explain the idea behind each building block and explain how you can use these to build your business. These are not difficult systems to build for a programming team. The idea behind each system is more important because it can be implemented in any programming language for any business.

Chapter 4.1

The Custom Field System



At some point in your application's lifetime, someone will want to add data that is not part of your data model. You'll be faced with a choice. Do you add a weird column to your database, or is there a better way?

Adding a column to your application to handle this request is not a bad thing if you're a new company. In the early days of a company, you must move quickly to keep up with customer demands. Sometimes you'll be in need of a quick short-term solution, and this will work "good enough". I've done this plenty of times myself.

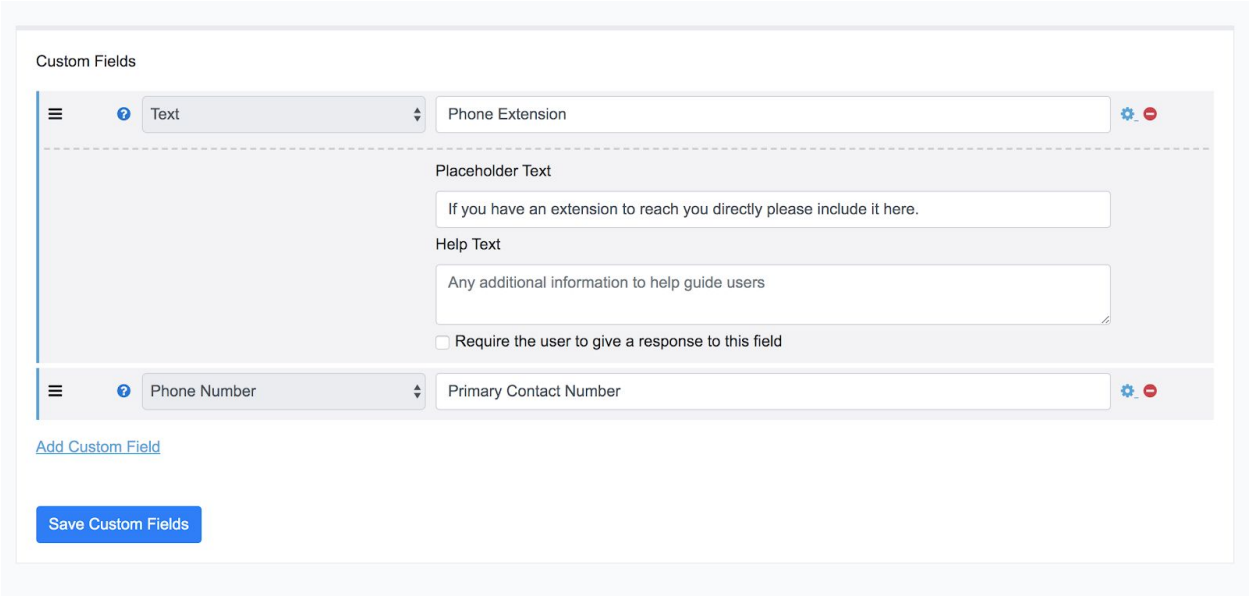
However, when this happens, in the back of your mind you should be ready for the next customer to ask for the same thing for their specific data. It's like having cockroaches... When you see one, that means there are ten more that you don't see behind your walls. Like cockroaches, if one customer is requesting this functionality, you should anticipate more requests on the way.

The nice thing is that if you handle the initial request as a new database column, you can always go back in the future and "migrate" that column and data into a new Custom Field when you have the time to build this system out. In fact, that's what I did recently with a start-up company that I'm working on. We have an "Organization" model where we added social links to the database as columns (such as twitter_url). Now that we have a robust Custom Field system, I will migrate that data into our CustomField system, and drop those database columns.

Your company won't be new and small forever! As you grow and get more customers, you need to think about the long term. The long term solution here is to have a Custom Field System.

What Does a Custom Field System Look Like?

I have worked on a handful of applications that have built Custom Field systems into their applications. Now that I have seen a few, there are things that all of them have in common. Here is a screenshot of what it might look like to add a new custom field:



This is a screenshot of the UI where someone would add, remove, or modify custom fields for this application. As you can see, the fields have the following information:

1. A "type" that describes the data type that the field will store. This can be a primitive, such as: integer, string, text, etc. Or, this can be rich data such as: email address, phone number, etc. This is a design decision you will have to decide.
2. Each Custom Field must have a name that is unique. I would recommend also storing a separate field on your model as a way to refer to the field through Liquid programmatically (similar to a permalink column that doesn't change).
3. Something that is not shown in this image, but that is important is that the CustomField must belong to something in your application. This will be a Polymorphic relationship. For example, the fields in the picture for "Phone Extension" and "Primary Contact Number" would likely belong to the user model.

Think of Custom Fields as like a database column, but able to be added, removed, or modified on the fly by configuration people. Going back to the fields in the screenshot above, let's say that we had a User model that had an ID, email, and name columns.

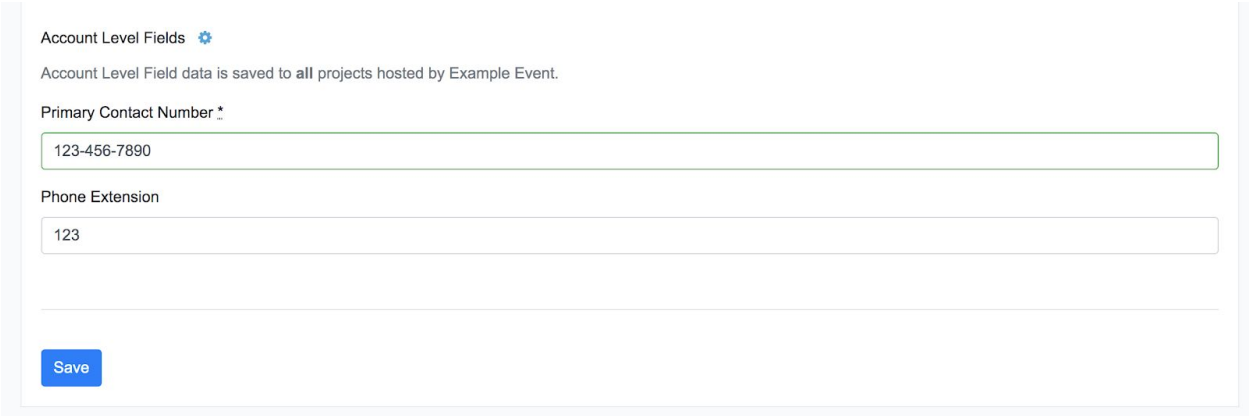
We are essentially saying that we are adding two additional fields to this model to store a Phone extension and a Primary Contact Number.

In the systems that I've seen and worked with, Custom Fields are typically separated by the Customer. For example, if Customer A is working with volunteers, then it is probably highly valuable to be able to store phone numbers in our system. If Customer B is working with online video gamers, then it is probably more relevant for them to store screen names as their way of contacting people. But that's the beauty of a Custom Field system. Each customer can decide for themselves what is important, and create the fields that they need.

In Custom Field systems, you typically want to enforce that the data type should never change. Or rather, once there are values, it can be extremely challenging to change. Thinking through this, let's say that you have a CustomField created to store strings (allowing characters such as a', 'b', 'c', etc). Then you decide that you made a mistake and want to change the value to store integers (1,2,3, etc). How are you going to migrate the data? There are some cases where it might be possible, and others where you will definitely have to use some creativity. In the system I've worked with, we simply don't allow the user to do this.

Custom Field Values

Now that you have your Custom Fields Setup, what's next? You have basically added on to your database schema without touching your database! The next step is to allow your users to enter data for these new "virtual" columns. Here is an example of what this may look like in your UI:



The UI must render the field depending on the data type. For example, if you have a Date field, you must give the user a way to enter a date value (with a date picker).

Use a text database column to store the data the user enters into the field, and then simply cast the value to the expected data type when reading the data. In some cases this won't work if your data type is too unique. However, in my experience, this approach does work well the majority of the time.

Validation

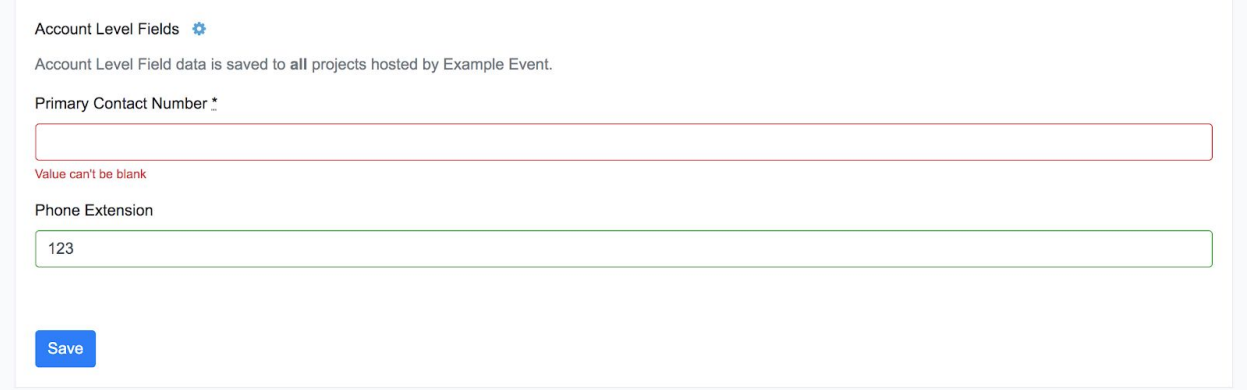
When you're setting up your Custom Field, this is also a good place to add validations to ensure the data you're collecting is correct. I've worked in systems that have two unique approaches toward solving this problem.

The first method of validating is to keep your data types to the bare minimum (primitives), such as "String". Let's say that you were trying to model a phone number. The way you would achieve this is by adding individual validations to your field to define what a valid phone number is. As a basic example, you might add a Regex validation to ensure you have ten numerical digits.

The second method of validation is to use "rich" data types with built in validations. For example, if you allow your users to select "phone number" as a data type, you can add built-in validations. The nice thing about this method to solving the problem is that it can be a little more user friendly in the front end when your customers don't

fully understand validations. However, the drawback could be that they want to remove validations that are built in.

A hybrid approach to the validation problem also works well. You can have rich data types, but also allow the user to add or remove additional validations.



You should treat Custom Fields the same way as any other data attribute of your models and validate them the same way. In a Ruby on Rails application, you want to validate your Custom Field Value data at the same time the model that is being saved is validating its own data. This way, you can handle when things are invalid in the same way.

When to Use a Custom Field Vs. a Database Column

One of the problems that happens with having a Custom Field system is that you will constantly need to decide what should be a Custom Field and what should be an actual database column?

Honestly, this shouldn't be an extremely tough question because it is a highly reversible decision. If you decide to make it a database column and you were wrong, it's easy to fix. If you decide to make it a Custom Field and you were wrong, it's just as easy to fix later on.

With that said, here are a couple things to consider when making this decision.

First, if you need to access the data through code, I would always recommend using a database column. Things get extremely weird and fragile when you start referencing data in your codebase (Trust me. I've done this before.) Going down this path will likely lead to other fragile code in your codebase, such as if statements to check what customer your code is processing (since most Custom Fields will be scoped by customer). If you can anticipate using the field for more than just reading it, then I would go with a database column or another solution.

Another way to see if you've made the right choice in retrospect is to look at patterns in your customer's data. Does every customer you have always add a "phone number" field to the user model? If that's the case, then that is probably an indication that this should be a real database column.

The data can also speak in reverse as well. Do you have columns on your models that only a couple of your customers are using? If that's the case, that is a good candidate for moving to Custom Fields. The great advantage here is that you're not cluttering the UI for every customer at the request of a small number of customers with the need for those fields. You've got a Custom Field system, so use it!

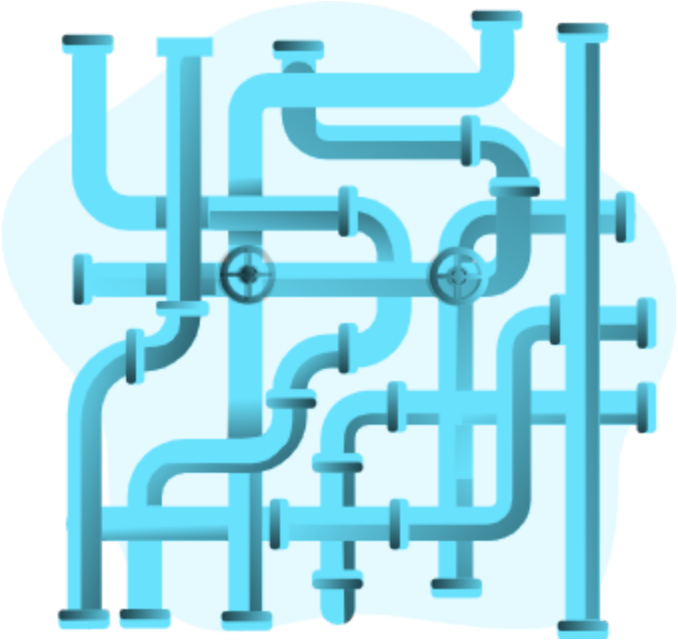
Custom Field Final Thoughts

The first building block of our configuration tool belt is Custom Fields. A good Custom Field system can save you from cluttering your codebase with customer-specific fields that no one else uses.

This is laying the foundation for storing any kind of data that your customer could require, and we will build on this foundation with the next 3 building blocks. Next up, let's talk about building a Condition System.

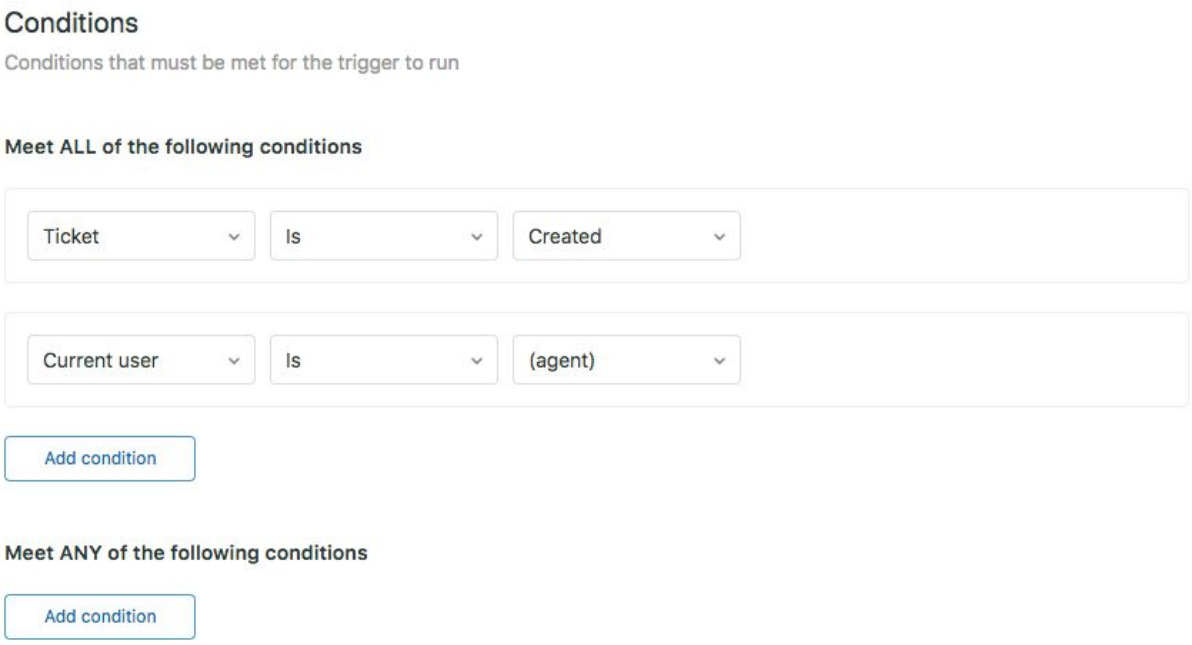
Chapter 4.2

The Conditions System



Now that we have our Custom Field system that is capable of storing any customer data we could need, the next building block we should think about is a Condition System. The basic idea of this system is to move customer data-driven logic out of your application code and into the data of our application. That may sound scary, but in reality it's pretty simple.

Just like the Custom Field System, many applications in the wild that must deal with complex use-cases have this kind of tool built in. Doing a quick Google search resulted in Zendesk's Condition System.



Zendesk's condition system is extremely similar to what I've worked with in my career. Each condition has a left side, an operator, and a right side. Individually, the value of the left side is computed, the right side is computed, and then the computed values are compared using the operator. Each condition either returns a true or false boolean value.

Individual conditions are combined together using a "Meets ALL Conditions" (a logical AND), or "Meets ANY conditions" (a logical OR). The result of the entire evaluation results in either a true or false boolean value. You can use this evaluation to make decisions in your codebase based on arbitrary business logic without ever polluting your actual codebase with this logic.

Example Usage

Let's look at an example using the scenario introduced earlier in the book. As a reminder, we have special code that must be executed if the order was placed for Michigan. Our code might originally look something like this:

```
if order.is_in_michigan?
  # ...
end
```

As we discussed earlier, this is not ideal since we have this customer-specific logic in our codebase. It's not bad by itself, but imagine if we did this 1,000 times. Things could get extremely messy. Fortunately, there is a better way! Let's see what this would look like with the Custom Field and Conditions Systems.

First, we would add a new Custom Field for this customer attached to our order system. That record may look something like this:

Data Type	Boolean
Name	Is In Michigan?
Attached To	Orders

Now that we have our custom field, we can set each Order that is placed to have an “Is In Michigan?” value of either true or false. Then, we would set up our condition as something that looks like this:

Order Is In Michigan?	IS	true

Under the hood, this would be using Liquid on the left and right sides (even though the right side is a constant “true” value, it should still be evaluated through Liquid). The Liquid representation of this condition would look like this:

{{ order.is_in_michigan? }}	==	true

Now, our Conditions System can evaluate this. It will start with the left value, which let’s assume that when evaluated by Liquid will return a value of “true”. Then, we can evaluate the right side of the condition. Since there is nothing dynamic about it, it will return a value of “true”. Then, the application can compare the results using the middle operator. Since “true” == “true” is true, our condition evaluation results in a true value.

Going back to our code example, now we can change that code to be something like this:

```
if order.conditions_pass?
  # ...
end
```

When the `conditions_pass?` method (or function) runs, it evaluates the conditions that store the business logic that has been configured. As you can see now, our messy customer-specific logic is no longer in our code base and instead lives as data in our database.

In the future, let's say that this customer has another crazy request. They now want the special functionality only if the person's name is "Adam" and the person's employee_id is "25B". You can instantly see how the customer specific logic would have grown over time. With our Custom Field and Conditions Systems, this ask is trivial.

First, we add our new Custom Fields:

User's Name Custom Field (In many systems, this may already be modeled in the database)

Data Type	String
Name	Name
Attached To	User

User's Employee ID Custom Field

Data Type	String
Name	Employee ID
Attached To	User

Now, we'll add our new conditions to the existing Conditions. This will end up looking like this:

Meet ALL of the Following Conditions

Order Is In Michigan?	IS	true
Order User Name	IS	Adam
Order User Employee ID	IS	25B

Under the hood, we are still using Liquid:

{{ order.is_in_michigan? }}	==	true
{{ order.user.name }}	==	Adam
{{ order.user.employee_id }}	==	25B

Guess what? Our code doesn't change at all! That's the power of the conditions system. You can introduce new customer specific business logic without changing any code. Now, when our "conditions_pass?" method is called, it will evaluate the new logic and behave accordingly.

This is extremely powerful because it gives your application the ability to be highly adaptable to customer's logic without becoming an unmaintainable mess. Going back to our original goal, customer logic can be configured by a configuration team, rather than the software engineering team.

What About No Conditions?

When you go to evaluate your conditions, what happens if there isn't anything there? This is an important decision to make because it will impact how your application behaves. Personally, I think that there are advantages to making conditions return true when they don't exist. This means that when the configuration team doesn't add any condition, behavior application-wide will be enabled by default.

You can use Conditions to control this in cases where you must disable functionality. In the worst case, if the configuration team needs to disable a feature for a customer, they can enter a condition that is never true, such as this condition.
Meets ALL Conditions

1	IS	2

Since 1 is never equal to 2, the condition will always fail and the functionality will be disabled.

Short Circuiting

Since the conditions System is using logical operations, there is room here to get clever about how your system handles evaluation. One way that I've dramatically improved the performance of a condition system in the past has been to implement short circuiting.

If you're unfamiliar with how short circuiting works, here is an example. Let's say that I tell you, I will give you $100 if "The sky is purple" AND "the day is Tuesday" AND "Your name is Bob". Well, since we are doing an AND operation, as soon as you hear "The sky is purple", it doesn't matter what the rest of the conditions are, the entire statement will always be false since the sky isn't purple.

This works in the opposite way for OR statements. With OR statements, as soon as a single condition is true, then the entire statement is true. Going back to the same example, let's say I will give you $100 if "The sky is blue" OR "your name is Bob" or "The day is Tuesday". Well, in this example, I will owe you $100 after the first condition since the sky is blue.

All production programming languages take advantage of this. You can too! This is a great way to improve the performance of your application.

Condition Weights

I remember one time solving a bug when we had a condition (one of many) that would be way more computationally expensive than all the rest. Our team had optimized the performance as much as possible under the hood, but there was only so much that

could be done. The work that was being computed was always going to be more expensive than other operations.

When you have short circuiting, you can be clever about ordering your condition evaluation. Let's say that you have the following conditions (with weight added in the third column).

Meets ALL Conditions

Condition 1	==	true	cheap
Condition 2	==	true	expensive
Condition 3	==	true	medium

It is in the best interest of your application to evaluate these conditions in the following order if possible.

Condition 1	==	true	cheap
Condition 3	==	true	medium
Condition 2	==	true	expensive

The reason this is the ideal order is because if the cheap condition fails, then we may short circuit and know that the entire result will be false without ever actually doing the work to evaluate Condition 2 or 3, which are the most expensive. This performance gain may be trivial for a single condition. However, in applications with thousands of conditions, this performance improvement can really add up to have an impact.

In reality, this can be extremely hard to pull off. It's often difficult to know how expensive an operation will be beforehand. However, this is just one example of how you can use a simple Logic hack to improve your application!

Chapter 4.3

The Rule System



The Third Building Block that we will talk about continues to build on the previous two. We have given our users the ability to arbitrarily add customer specific data and logic to our application. Now, we will also allow our users to take action within our application at certain key moments.

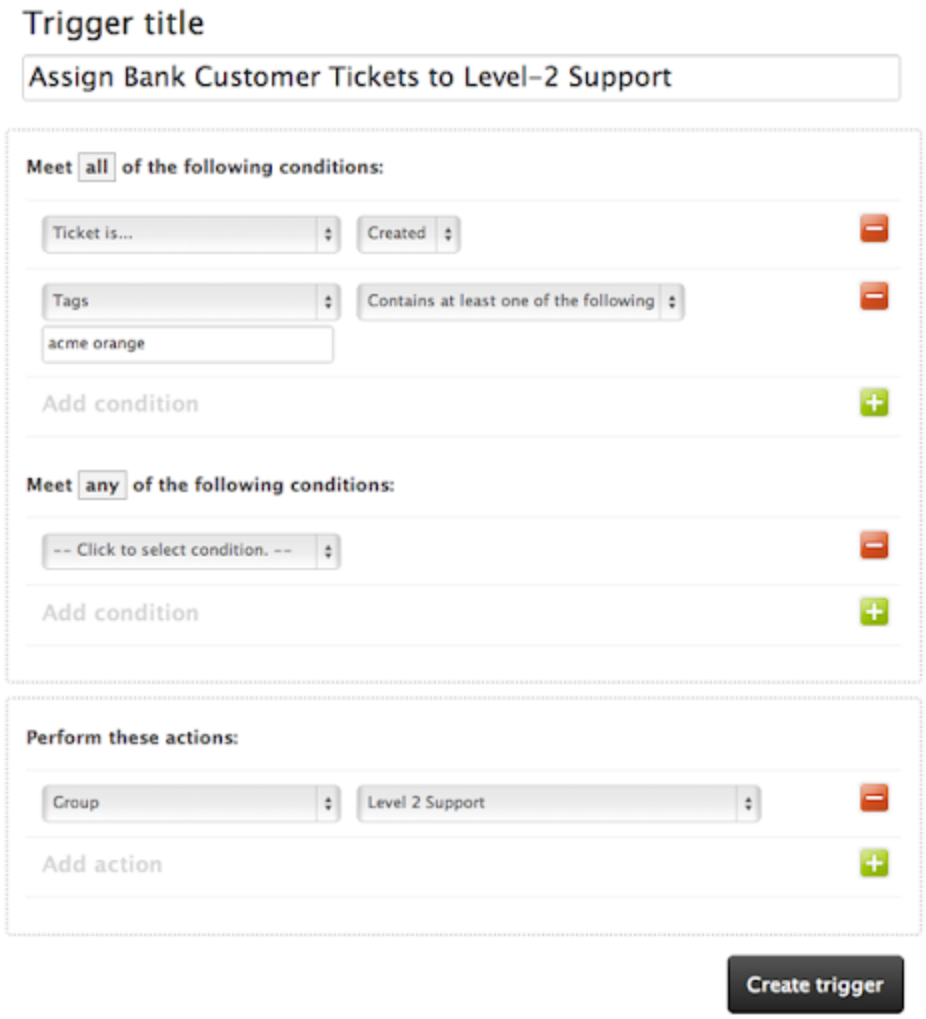
The third building block we are discussing is called the Rule System. For programmers, this is equivalent to a “pub sub” system in programming languages. For web developers, the most common example you would be familiar with is working with Javascript.

```
<body onload="function(){ alert('testing'); }">
```

Take a look at this example. We are waiting for an event called “load” from the body of the HTML document, and when this happens we want the function to be executed. As you’re probably aware, in javascript you can add as many functions as you would like to be executed when the document body loads.

In this example, we have one “publisher”, which is the body who would broadcast the event “load” at the right time. We also have many “subscribers”, which are actions that happen when the event is broadcasted. This is an extremely important concept in programming, and exactly what we’ll be modeling with the Rule system so that configuration users can take advantage of this third tool.

Let’s take another look at an example Rule system using Zendesk’s as our example.



You can, right off the bat, see how we are reusing the Custom Field and Condition System. We are now adding two more elements. We are adding a "Trigger", which is our event that will be broadcasted within the application. We are also adding an "Action" to configure what should be done when the event is broadcasted.

Let's take a look at an example. Let's say that our customer wants a custom email to be sent to all new users that sign up to the application if the user's favorite color is blue. Thinking back to the last two chapters, first we'll create a new Custom Field to store the user's favorite color, and then we'll set up a condition that compares their favorite color to "blue".

Now, let's create our rule. In pseudo code, our logic will basically say this:

When **a user is created**, if conditions are true then **send custom email**

In our codebase, let's look at an example of how you might trigger this rule in the User model:

```
class User
 after_commit :trigger_rule, on: :create

 def trigger_rule
  Rule.trigger("user_created", self)
 end
end
```

This is basically our application broadcasting "A user has just been created" message. A rule that matches the "user is created" trigger, would then execute. It's processing code might look something like this:

1. Verify the new user is for the same Customer as the Rule
2. Check if conditions pass (arbitrarily configured business logic)
3. If conditions pass, then do the action

Similarly to the HTML example, you can also create many Rules for the same broadcast. Let's say that in addition to the custom email, we also want to send the customer a custom SMS Message. We can do that easily. Let's simply add an additional Rule:

When **a user is created**, if conditions are true then **send custom email**

AND

When **a user is created**, if conditions are true then **send custom text**

You can add as many rules as you would like to be triggered. One thing to note, I would suggest that you process these rules in a background job on your server. When you add many rules, things can become computationally expensive. You don't want your user waiting when they are signing up for all this stuff to happen!

The Rule System Final Thoughts

Similarly to the first two building blocks, the ultimate goal is to move the responsibilities of software engineers to the configuration team. The Rule system takes things one step further by actually allowing the configurers to orchestrate basic workflows.

Before, a customer request such as “Can we send a tweet every time an order is successful” can be a difficult and expensive request to fulfill by software engineering. Now, all the software engineering team needs to do is add a new action to the Rule System. Once that is completed, every customer going forward can be taken care of by the configuration team if they need something to be tweeted.

Over time, you will create many “hooks” (or triggers) throughout your application so that the configuration team can add more functionality at the right time for customers. Once again, the great thing about this system is that once you have a good library of actions, the configuration team can take care of many customers without the developers ever hearing about those requirements. Functionality can be added, removed and modified through the rule system without ever changing a single line of code for existing actions and triggers.

Chapter 4.4

Email Template System



With an email template system, you can give your users complete control over the email content sent by your application. Using Liquid, your users can use dynamic content within their email's body, which is sometimes called "Merge tags" by the large email marketing companies.

Out of all of the uses I've seen with Liquid, this is by far the most popular. Lots of applications that only send emails have this type of system since this is the experience their users expect.

There are a lot of benefits for your application to have its own Email Template System.

- Allows your users to customize the default emails that are being sent from your application. This can include branding, personalized copy, etc. As a software engineer myself, I have vastly underestimated the value of personalizing emails in the past. Emails are important. Customers that I have worked with care highly about what is being sent to their users and how often.

- Increases the value perception of your company's product. At one of my old jobs, we used to ask ourselves "How can we make this product look expensive?". Adding more functionality to your application is one way to help your customers see more value.

- Makes your emails dynamic using Liquid! Emails that are personalized have a higher chance of being read and for users to take the action that you want. Users can customize their emails using the same Liquid tags that they're already used to since it is used throughout your application. They're not learning an additional system.

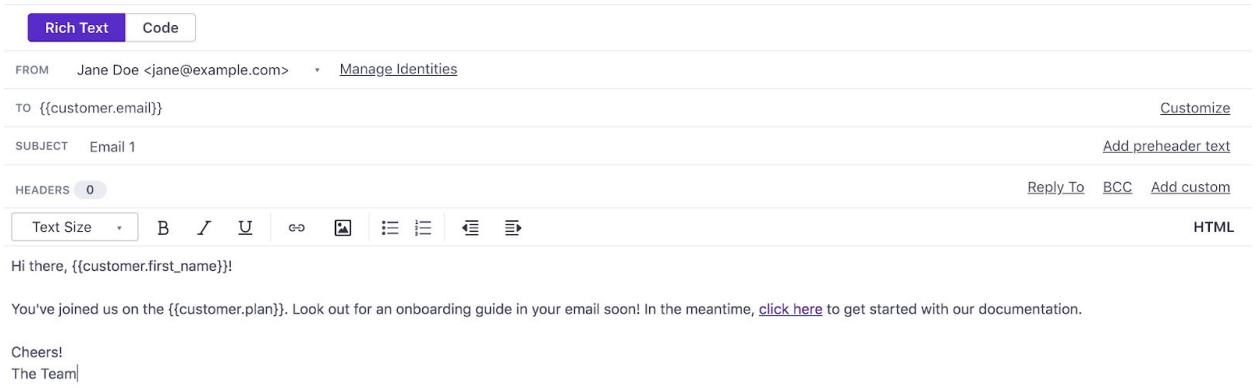
- Have complete control over your email system. If things go wrong, *and that never happens*, it's good to have complete control over your system so that you can address anything customers are unhappy about. I've gotten some weird requests in the past that without complete control would be hard to fulfill. At the end of the day, you need to keep your customers happy after all.

One *huge* advantage to having this system is that it will work with the other three building blocks. Think of the Rule system. With an Email Template system, now you can have an "Email" action that automatically works with Custom Fields and Conditions. If your customer wants a custom email to be sent at any time, this is instantly possible. The data will flow through the Rule system and be used to render the email that is sent.

Ultimately, this final building block achieves the same goal as the previous three. A clear separation of responsibility between your software engineering team and the configuration team. How many times has your engineering team gone into emails to make minor copy changes? You don't want your expensive engineers doing this kind of work! Give your configuration team the tools that they need to make customer changes themselves.

Example Email Template System

Here is an example of what an email template system may look like:



Your email template editor should have a few fields (probably exactly what you would expect):

- A “From” field, which is the email address where emails from your system will be sent from. Keep in mind, you must send from your own domain name, unless you have specifically configured other domain names with DNS settings.
- A “to” field where the email address to send to will be populated.
- Subject and email body content. Using a WYSIWYG editor for your email content is a great idea and will allow your team to build some pretty incredible emails. I’ve always been surprised at what you can pull off when you have some time to put into it.

My recommendation for your system is that every field *should* allow for Liquid to be entered. When the application sends the email, every field can be parsed and rendered with Liquid, even if no dynamic variables were entered (or even if you don’t think anyone will ever need Liquid in that field). It can be an abstract thing to wrap your mind around at first, but even a “constant” string with no dynamic variables can still be rendered as Liquid. For example, in the string:

“Hello Adam,”

There is no dynamic content with {{ }} tags. However, you can still render this using Liquid, which would do nothing and return the same text. This is a nice hack to keep in mind. Rendering with Liquid is extremely fast, and it is always a nice surprise when you’re trying to solve a weird customer ask in the future and things just work because you planned ahead and made everything work the same way.

A “War” Story

When I think about Email Template editors, I always think back to an instance when there was an upset customer because their email didn’t look exactly how they wanted.

This customer was a “whale” for us, so obviously it was important to keep them happy.

The customer had an image banner across the top of their email, but they didn’t like how wide it was. They demanded that it was exactly 6 inches! So, our configuration team went into the email editor, got their ruler out from their desk drawer, and made the change while holding their ruler up to the monitor to measure it out perfectly. Our team had a laugh and the customer was happy.

Beyond Email Templates

This last building block, the Email Template System, is just one example of what is possible with Liquid now that you have the Custom Field, Condition, and Rule systems. The possibilities are truly endless about what actions you can allow your configuration team to have, although I would recommend you base your actions and roadmap on customer demand. Keep in mind, Liquid can make **any** document be data driven. Some obvious other actions to build out may include: SMS capabilities, Twitter, HTTP Web Requests, etc.

When you make decisions in a way where you’re dividing responsibilities among your team, you’re building for the long-term. It might be more work and effort up front, but your effort will pay off when a new customer is onboarded and the development team doesn’t have to lift a finger. At my last job, we called this “Having exponential impact”. When a new customer comes on board and you don’t do anything, it’s not because you’re lazy. It’s because you already did that work, and now your code is working for you.

Chapter 5

Using the Building Blocks



Now that we've discussed the four building blocks, let's walk through an example of how these tools can be used to onboard a new customer in an example scenario.

For this example, let's say that we have an application where employees go to do training for their company. So, this is a B2B business where our customers are companies that need to train their employees. Our users are the employees of those customers who need to complete their lessons.

Now, let's imagine that we have a new customer that is onboarding. This customer has the following requirements:

- There are employees that report to different offices depending on their location. Lessons must be customized depending on the office location of the employee.

- Employees should receive a customized email when they sign in to the application for the first time. The customer wants the email to be branded and personalized to help with getting employees to *actually* do the training.

- If the employee works in the field, send the employee's manager an email when lessons are completed. That way, the manager can issue any equipment the employee needs.

Let's look at the work involved with this customer onboarding. We have:

- Three new pieces of customer-specific data we must store on the user: Their office location, their manager's email address, and if they work in the field or not.
- We have one new business logic driven decision that must be made. We will enable additional functionality If the employee works in the field.
- There are two events that must be broadcasted by our application when they happen so that a Rule can be triggered (and subsequently the Rule's action)

- We have two new custom emails that must be created on behalf of the customer.

Without the four building block systems, this would be a lot of work and involve the software engineering team adding many customer-specific database columns and logic. Luckily, we have our Custom Field, Condition, Rule and Email Template systems.

Let's assume that our application already triggers a Rule broadcast for when a new user signs up (because a previous customer already requested that feature). However, no one had ever requested the Rule system to trigger when a lesson is completed before. So, that becomes the work involved in this customer onboarding for the software engineering team. This would result in a single software story to add a Rule trigger when a lesson is completed.

Now, while the engineering team is working, the configuration team can also get started. First, they would create three new Fields for our customer:

Office Location

Data Type	String
Name	Office Location
Attached To	User

Manager's Email Address

Data Type	String
Name	Managers Email Address
Attached To	User

Works In Field?

Data Type	Boolean
Name	Works in Field
Attached To	User

Once our custom fields have been created, the next step for our configuration team would be to create the two custom emails that need to be sent for this new customer. They can use the Email Template system to design two new emails to be sent from our application to the exact customer specifications without ever leaving our application.

Then, we will set up our first Rule, which can be expressed in pseudo code as:

When **a user is created**, if conditions are true then **send custom email**

Finally, once the engineering team has completed their work of adding the new Rule Trigger, the configuration team can add the second rule we need, which can be expressed in pseudo code:

When **a lesson is finished**, if conditions are true then **send custom email**

Our second Rule should only be sent if certain conditions are met. In this case, we only want the email to be sent if the employee works in the field so that their manager knows to send them the equipment that they will need. The condition we add would look something like this:

User Works in Field	IS	true

Or, under the hood, it would be stored in Liquid as:

{{ user.works_in_field }}	==	true

Now that we have this condition added to our Rule, we will only send emails to the person's manager if their "works_in_field" value returns true.

With our second Rule in place, we have fulfilled the requirements of our newest customer! The software engineering team had one story to work on, and the configuration team had three fields, two Rules, one condition, and two emails to write.

This is the perfect example of how two different teams within your organization can work together to make a customer successful for the lowest cost to the business possible. Teams can work in parallel to onboard the customer as quickly as possible with the engineering team working on the Rule at the same time the configuration team is configuring things. Ideally, this request would be fast for your engineering team to build so that team can move on to the next high impact feature for your business!

Conclusion



My goal with this book is to help you use Liquid along with four basic building blocks to help your companies' teams work better together, fulfill customer's requests more efficiently, be more flexible to customer's needs, and ultimately make your business more money.

Impact Vs Effort

In my last job as an engineering team lead, one of the things we constantly stressed to our team was the difference between impact and effort (Being busy versus being productive). It is possible to put in a lot of effort and have very little impact. Inversely, the opposite is also possible. You can put in very little effort and have a huge impact.

There are tons of examples of the difference you see all the time if you learn to spot the difference and pay attention. Here are few:

1. **High Effort, Low Impact:** The "Juicero" Silicon valley startup raised $120M to create a $400 juicing machine. The company was ridiculed when users found out you could squeeze the juice out of the packet without ever needing the machine at all.

2. **High Effort, Low Impact:** The Movie "John Carter" had a production cost of $263M, but lost $122M. There are lots of examples of "movie flops" you can find by Googling.

3. **Low Effort, High Impact:** The game "Flappy Bird" was created by a single developer in 2-3 days. At the height of success, the game was making $50K per day.

4. **Low Effort, High Impact:** The movie "Paranormal Activity" was created on a budget of $15k, and went on to make $193M worldwide. Blumhouse Studio has had several low cost movies with this type of success.

Is your team working with low effort and high impact?

Get the Highest Impact From Your Team

Your team should be focused on building features that have the lowest cost to the company and the highest levels of return possible. That's the beauty of software. A one time up-front investment can pay off for years when things are built correctly.

I would urge your team to explore the four building blocks discussed in this book, and look at how Liquid can help in other areas of your business as well. Incredible profits are possible when your engineering team is working with this mindset.