Chapter 9 | The First Minute

With our internet machine, more goes on in the first minute than in any other minute. The user does not experience this – well except for the resultant flurry of email. A web-based application, also called software-as-a-service, relies on external services, and runs more lines of code during the first seconds as the user's payment is processed and the user's new services get built.

In Chapter 2, I defined "The Cloud" as the mathematical center of the internet. This definition, while not-right is also not-wrong. The importance of the definition stems from the amount of time it takes to communicate with a vendor, or partner, or service provider at the Center of the Internet. The data exchanges ought to be blindingly fast and achieve a high degree of reliability. In the old days of computing, even a decade ago, software developers did not trust that a partner would respond with data fast enough nor consistently enough. Therefore, we architected redundancy and measures to protect our own operations from their slowness or their failures. We've moved forward. This more modern technique approximates the definition of an "internet machine". Modern web-based software has been constructed within the fabric of the internet.

In Chapter 3, I explored software especially software-as-a-service as the tool chest of the modern economy. I tried to break barriers between software, engines, machines – they are basically things that does stuff. In the case of Podcast Flow, it is a suite of software that operates together. PodcastFlow, the application, aids podcasters with the planning, promotion, production, and profiting from podcasting. We also offer courses through our learning management system: Podcast Plan in a Weekend, Infinite Interviews, and Guests to Grow.

I have been stepping through challenges we faced with collecting subscription fees, collecting sales tax, and establishing a unified and secure means of granting users access to our machine – the suite of software. I ranted about the United States Supreme Court's decision called "South Dakota v Wayfair Inc." – which resulted in every internet business who is based in the United States and sells to customers in the United States to collect sales tax for the thousands of tiny sales tax jurisdictions.

Step by step I demonstrated the building blocks of a complex process. And I discussed a lovely failure we had with PayPal – and a waste of time and money.

The first minute of interaction with our machine resembles actions needed for nearly every "internet machine" from Netflix and Hulu to your favorite dating apps or whatever other subscription type software you imbibe.

The first minute must meet objectives for three entirely separate audiences. In short, there are three bosses involved simultaneously. The first boss, of course, are the technical people. The other first boss are the marketing people. The other other boss is the user.

Each of these three bosses expect different behavior, different actions, and must be served equally.

I do put the technical teams’ needs ahead of the others only because nothing works if the tech fails. I also put the technical teams’ needs first because it is **my** perspective. I think in terms of building blocks – software design and development resemble a physical architectural process in my mind. I often try to use Lego as an illustration, but I admit it doesn’t work every well.

The classic Lego block, say the ole 2 by 4, has smooth edges. From the outside it appears its job is to hold up the block or blocks above. Yet, with colors and blocks of many shapes, amazing structures appear – even machines that whir and move. It takes a lot of blocks and each block must be in its place, well supported to be structural sound.

In my programmer’s mind, that humble block can substitute for something else – it does not have to be a block of six sides and smooth edges. It can be its own tiny machine doing it own tiny machine things. Remember my definition of machine: a thing that does stuff to something.

Maybe you think of a black-box, or a magic box – or even one of those smart speakers in your house, or your mobile phone. Maybe you can think of the computer on the original Star Trek TV show – Spock asks “Compute to the last digit the value of Pi”.

[PLAY CLIP]

That’s not a great example as the computer does not actually return a value. Software architecture depends on discrete building blocks that each does their own thing yet forms a solid structure.

Recurly, introduced in Chapter 6, is such a building block. It does all sort of black-box or magic-box things. We don’t have to build it. We only have to know ho w it behaves. We need to know how to talk to it and how to listen to it. It connects to our system. It shares data with our system.

What do we need to do during this first minute? I require about 1 minute to read this list.

1) Accept the payment for a new subscription
2) Create or update the customer information
3) Create or update user information
4) Update the security profile: who has access to what products and for how long
5) Setup the user account at Okta
6) Assign the user to one or more Okta groups – granting them access to the learning management system or PodcastFlow, the podcast management system
7) Update email lists with information about what someone bought and when
8) Get user credential to be emailed to the user
9) Get a receipt emailed to the user
10) Get a welcome email sent to the user
11) And whether they bought just a course, or the full suite, we setup their first podcast show in Podcastflow

12) Calculate their sales tax, if applicable
13) Track their sales tax information with a previously unmentioned vendor called Avalara. This vendor's tool is bolted onto the side of Recurly.
14) Get the customer data sent to the email marketing tool and associated with the product they bought

Done.

The new users will get at least three email from us. Recurly will send a receipt. Okta will send login instructions. And PodcastFlow sends a welcome email.

Our staff get a few email too. We get a buy alert. We get email from various points in the credit card transaction process. Recurly processes the subscription. That gets handed to another service who processes the payment on behalf of our bank. At the end of a business day, we get a daily reconciliation of transactions. The next day, the values appear in the bank account as payments.

It sounds like a lot. And there is a lot of complexity. If your experience is that of a user buying something on-line, this maybe your first look behind the scenes. If you are a programmer listening to me, many of these steps will be familiar.

Let's start with Recurly. We have a website hosted by Recurly. The Recurly logo is visible on the lower right corner of the screen. In the URL at the top of the page, one would observe the icon of a lock that is snapped shut – locked. The locked icon informs us that the link between the user and Recurly, the host, is encrypted with certificates conferring trust. In that address, it includes "recurly.com". On this page is our product or products and descriptions.

We like having their name on this page. It clearly informs the user that PodcastFlow is not collecting payment data. By hosting in conjunction with Recurly, we reduce risk because we are not storing critical and confidential financial data: no credit card numbers, etc. They manage coupons, discounts, renewals, and all of that. They possess the required expertise with these skills.

The customer arrived at this page from a sales process that expounded with great enthusiasm on the value of the product or service. The customer clicked through to our buy-page which closely resembles a check-out page for a lot of on-line marketplaces – an added value to us. People trust the familiar.

At the conclusion of the purchase, Recurly has all of the data that every other system needs and all three of the bosses want (tech folks, marketing folks, and buyer/the consumer). Recurly sends a polite email to the buyer as a receipt of the purchase. In fact, we have switches to configure what email Recurly sends on our behalf. We ask them to email customers with a new subscription, a change in a subscription, an invoice, and/or payment confirmation.

We prefer this confirmation coming from them as a means of enforcing trust. Our logo and name are also present, we inform customers and the world we are not collecting confidential financial data.

Buried within the Recurly process is a sales tax calculation. A firm called Avalara looks at the type of product, in our case: software-as-a-service, or subscription software, same thing. They need to know if this service or product is taxable in the user's jurisdiction. I think they use the postal code provided by the user. If the purchase is taxable, the amount of the tax must be displayed to the user before she, or he, checks out. That is an important disclosure.

The systems we manage at PodcastFlow do not need to directly interface with Avalara. They've done their job when they calculated and tracked the sales tax. Thank you, Avalara. So far, we have also not paid any sales tax to anyone. And yet, it costs us a fair amount of money. Oh well, cost of doing business on the right side of the law.

How do these data move from Recurly to PodcastFlow?

The building blocks of software only work if they can process and move data along. Snapping pair of Legos together seems intuitive and obvious.

Snapping Recurly into PodcastFlow? No, not obvious.

First question… Does Recurly send the data to PodcastFlow? Or does PodcastFlow go fetch the data from Recurly? Both are possible. Which is best?

Only Recurly knows that it is a new subscriber. Therefore, it makes sense that Recurly shout out to PodcastFlow: "Hey, I have a new customer." They toss the data at us. This is called a webhook. We configure Recurly providing it with credentials and a destination address – which is exactly a website address same as a URL.

Recurly burbs data at us with any change. We capture all of that and store it. Someday, we may even figure out if we want it all. Just to be polite and for all sort of technical reasons we catch and store it *all*: new subscription, new customer, new invoice, new payment, new transaction, all of it.

We decided to respond only to a specific type of data. When you peel back the layers on these data packets, they tend to be similar. We picked the one that is most representational of activity we care about. We opted for a paid invoice. If we hear or see a "paid invoice" data set coming at us, we pay more attention. All the others get tossed in a figurative place called the bit-bin. Think trashcan or dustbin. Oh, sure we keep it and ignore it, both.

We unfold the lovely data set with the Paid Invoice. These data are in JSON, a slightly more modern format than what I first learned in the 1980s – the old Comma-Separated Value or CSV. JSON is a damned cool system for communicating data. It is an open and public standard. It is human-readable, and completely agnostic about the context: computer language, computer type, none of that matter. A verbal description will do little good, but the page in Wikipedia and the example there is worth looking at. JSON is spelt jay-ess-oh-en, Javascript Object Notation. I have clients through all of my career who knew and played with data they pulled from spreadsheets. Importing a comma-separated list into a spreadsheet is done so easily. For database folks, you can grow to hate the ole comma-separated list.

Here is a pile of neat data that works oh so occasionally. For its decades, and even its on-going popularity, it is rather fragile. A missed placed comma makes it a mess. In JSON each element of data has neat little label with it. Hey, I am a first name it says. Or Hey, I am a postal code. Data is real and complex and readable and useful and all sort of cool things.

The Paid-Invoice data from Recurly holds a fair amount of data including the Invoice number. Yes, it has the customer name, address, information about what the customer bought – all of the information one would expect on an invoice. Invoices are the same whether digital or paper.

We have two key of questions…

1) Is this a new customer or returning customer?
2) Is this a new product subscription or a renewal?

If the customer is a returning customer, we do not have to set up a new user. We do not have to setup the customer information within PodcastFlow and the Learning Management System, Maestro. Whereas, if a customer is new then yes, we have these additional steps.

Similarly, if a subscription is a renewal, then we have a simple step of updating the ending or expiry date. On the other hand, if the subscription is new, then we have several additional setup steps… new permissions, new access, etc.

I am not going to endeavor to describe a flow chart in a podcast – even printed most people hate them. Building an internet engine requires that we have little blocks that do stuff – little sub-engines… is that a thing?

One sub-engine maybe a building block that sets up a new customer.

One sub-engine maybe a building block that sets up a new subscription.

One sub-engine maybe a building block that setup up a new user.

You get the picture right.

Programming then becomes the process of asking the right questions: New or existing? If data are new, we insert them into our Oracle database. If existing, then we do a quick check of our data and update anything that has changed.

This engine we’ve built, that all programmers or tool-smiths build, involve smaller elements that operate independently. We can test them separately. They each do their own little machine-like thing. If we tell a little sub-engine to create a new user at Okta, it bleeps and whirs as we communicate with Okta setting up a new user. When that sub-engine is done, it give us an ugly and long Okta user ID number. The long Okta user ID number is about the same as saying: “Done”.

These little sub-engines operate entirely independently, Okta does not give one single hoot if a bill was paid at Recurly, nor that the sales tax was fussed over. The Okta sub-engine does Okta-like things. The Recurly sub-engine does Recurly things. Compartmentalized is a fair description of how these sub-engines work.

Our software steps through various decision points or branches as we call them when coding. Is this customer new? Yes or no. If yes, run the Create Customer sub-engine. If no, just get me the customer ID so I have it in my hand as I continue.

We call these baby-steps… Well I do anyway. I have a lot of saying when working with software developers and training them – and when troubleshooting my own stuff. They become shorthand for coordinating a team too. These phrases and the related way approaching problems becomes part of a team’s culture.

Let’s step backwards to the very first action when Recurly sends us data about a new invoice. This triggers the fourteen-actions that follow – that list that starts: update customer information, set up user, send emails, etc.

The very first time we play with this, we need to know that Recurly is sending data. One or two of us gather at a screen – we do this with a variety of collaboration tools. Our firm has never had a real office. Everyone has always worked from home.

We focus in on the process of having Recurly send data and we catch it. Failure is expected to the point where even if we are successful, we have doubts. It never works the first time, rarely works the second time. When you have it working, it should have that perfect reliability one sees with a light switch: switch on, light on; switch off, light off. The same coordinated response with every toggle of the switch. That reliability takes tries and failures – often deliberate failures.

In time, I ask: Are you walking the dog? Or is the dog walking you? I am asking who is the boss. Is the light switch actually controlling the light? Or is there another factor or set of factors in play?

A developer cannot move forward to the next step until they have perfect control and perfect reliability. You can't let the data, or the program, walk you around. The programmer must be the boss. The programmer must control one hundred percent of the actions – we do not like randomness in this process.

This is the most difficult lesson for a rookie developer. It becomes easy to see what data looks like and how to manage that data that you do see. Additionally, You must also accommodate what you do not see and the crazy variation nobody thought of. Know what we call these when they appear?

We calls ‘em bugs. “Yeah, sure it worked great for the first ten tries, then it failed”. What changed? Did the data change? Did the programming change?

Oracle has this absolutely lovely trap for programmers. Oracle manages null data with precision, the sort of geeky precision few humans live with. One of the data types in Oracle is called the boolean. The textbook will inform you that it stores the values TRUE or FALSE. Even according to Wikipedia: “the Boolean data type has one of two possible values (usually denoted as TRUE and FALSE)”. I do wish it were true. Oracle recognizes absence of data – also called null – as neither true nor false. Therefore, a boolean value, a data type that is supposed to have two states, has three: true, false, or null.

Same thing is true for numbers and all of their data types. It takes a while to understand that zero is not null. Null is the absence of data whereas zero is just another number on the number line.

What is one minus zero? One!

What is two minus zero? Two!

What is two minus two? Zero!

You get it, right? Here is the toughest lesson for the rookie:

What is one minus null? Null.

What is two minus null? Null.

What is null plus zero? Null.

Does one equal one? Yes, true!

Does one equal two? No, false!

Does one equal null? Null – neither true nor false.

You are a young programmer writing code and it works the first ten times perfectly. Then a number appears as a null. Does one equal null? Neither true nor false, but null. These little programming sub-engines fail. Life gets miserable, bosses get anxious, customers get frustrated. We have a bug on the software

Learning to anticipate the unexpected in the data is the only prevention. We have a bucket full of tools and fifty years of cool coding tricks to solve problems. But we can only solve problems that we see or that we anticipate.

Encountering null data is just one case when things go poorly -- that poor dog gets off the leash. The programmer needs to learn the difference between “walking the dog” or “being walked by the dog”. It can be hard to tell sometimes. Sometimes, as a programmer, you break what you fixed to prove you have control. You break something new to test resiliency. You have to learn to toss empty data and bad data at good code to see just how badly things will go.

“Baby steps” is not enough. A developer must learn to demonstrate and prove control. Remove a line of code or modify the data, see how it goes. The desire is to build structures that are both strong and accommodates weird stuff.

Once you feel great about things, you call this a new baseline. Then you take one baby step forward. As you build and work, you first strive to get stuff functional. Then you strive to break it and stress it. You break, then fix, then break, then fix. Like that light-switch trick. But instead of just toggling the same switch, you fuss at a different variable – change one wire, change the bulb, change the voltage. Make one change, and test. Get back to working, then make another change, a different change. Fix it. Lather, rinse, repeat.

It is fun. I have been at it since I was in grade school – pre-teen years, early teenage years.

Above, I detailed fourteen steps that need to take place within the first minute. It starts with a purchase at Recurly. Within milliseconds, we look at the data we got from a partner and trigger as many as 14 new steps with several vendors or partners. Data travels to and fro, jumping between the building blocks that we wrote and external services. Building blocks that must be solid, but flexible. Building blocks that form a foundation for the rest of a structure to stand on.

In the olden days of the internet, developers were hesitant to share data. The complexities involve security, and structuring the data, and who sends, and how often, and how reliable, and more. Of course, people shared data which is why standards such as HTTP and JSON came to be. Even with these standards, one also needed tremendous confidence in the reliability of transmitting data. I attended meetings when at FedEx with telephone companies local to the Oakland airport. We were building a small data center to support the international movement of air freight. We wanted a data circuit from each of two providers or vendors. We wanted the data circuits to make separate physical approaches to the building – a means of protecting operations from a digger on a street. Let’s make sure that one big yellow machine poking a hole in the earth does not take down both circuits.

In the late 1990s the internet did not have a center, there was no “cloud” as we think of it today. Data had to move from one edge or one facility to another over very real distances – taking time. Now we just push the data to the Center of the Internet, to The Cloud and work there. Absolutely, we pay fees to work there, but the vendors and services are milliseconds away with near perfect reliability. That makes creating a reliable, fast machine possible.

We can execute 14 steps across multiple vendors: Recurly, Okta, Avalara, Authorize.Net, TD Bank. The data are encrypted. Access is controlled, monitored, and regulated. The standards are open and public, but the process is secure and private. Within a minute, thousands of processes execute at the Center of the Internet, each taking milliseconds, each nearly invisible to the user.

The three bosses: the customer; the marketing team; and the technical team each get the information and services that they need.

The customer receives their goods or services. In our case, access to a class or access to software to help podcasting. The customer gets professional looking receipts and confirmation that their financial data was treated respectfully.

The marketing teams gets immediate confirmation of someone buying, what they bought, maybe even some clue as to why they bought and how they bought the product or service. The customer’s name and email get put into our email system so that regular care-and-feeding email can be sent on a regular basis. We ask how it goes… We ask if there are questions… We introduce ourselves… That sort of thing.

The technical team can relax a bit. The machine works. While that should not be a surprise, it often is. For those of us who have watched things go horribly wrong, watching a machine bleep and whir with little attention is a lovely feeling.

Nearly every senior technical person has a story involving sleeping on a conference room floor, or in a computer server room floor after a tiny mistake. At that same Oakland FedEx facility, I had a young staff member type: SHUTDOWN 0 0. He was showing a buddy what he learned. The servers shut down immediately. The databases corrupted. 2 vice presidents, three or four managing directors, several senior managers were all to be present for a major system test that day. And I sat at two keyboards with sweat dripping down my nose: one phone linked me to Australia for support and another phone linked me to Europe for support from another team. My boss, Beth, paced impatiently behind me. FedEx can equate seconds of delay with dollars of lost revenue. The shutdown command executed a few hours before the operational workday ruined us and delayed operations.

So long ago on the calendar, yet so fresh in my memory. It was not my first horrible mess-up with systems, servers, software, and operations. It was not my last. Several times in Iraq cuts in critical fiber optic connections prevented soldiers from accessing communications, cameras, and such. And even in Puerto Rico, I have a solid memory of my business partner John running down a street in San Juan to save a server before it crashed.

I prefer the calm and boring operations of an internet machine ticking along.

This week I got a paper bank statement – so retro, huh? For the first time, the document showed deposits generated by credit card purchases from customers buying our stuff. It is thrill – looking at daily deposits into a bank account after so much work. Yet in the back of my head are these memories of stupid mistakes: big yellow digging machines cutting fiber optic, and young staff members typing: SHUTDOWN.