The three kinds of platforms you meet on the Internet

- SEP 16, 2007

[Title with sincere apologies to Mitch Albom and his **wonderful book**.]

One of the hottest of hot topics these days is the topic of *Internet platforms*, or *platforms on the Internet*. Web services APIs (application programming interfaces), web services protocols like REST and SOAP, the new Facebook platform, Amazon's web services efforts including EC2 and S3, lots of new startups talking platform (including my own company, Ning)... well, "platform" is turning into a central theme of our industry and one that a lot of people want to think about and talk about.

However, *the concept of "platform" is also the focus of a swirling vortex of confusion* -- lots of platform-related concepts, many of them highly technical, bleeding together; lots of people harboring various incompatible mental images of what's about to happen in our industry as a consequence of various platforms. I think this confusion is due in part to the term "platform" being overloaded and being used to mean many different things, and in part because there truly are a lot of moving parts at play that intersect in fascinating but complex ways.

This post is my attempt to disentangle and examine the topic of "Internet platform" in detail. I will go at it by identifying *three distinct approaches* to providing an Internet platform, and project forward on where I think each of the three approaches will go. At best, I might be able to help make a new landscape clear. At worst, hopefully I can at least provide one framework for future discussion.

Let's start with a basic definition. From a **previous post**:

A "platform" is a system that can be programmed and therefore customized by outside developers -- users -- and in that way, adapted to countless needs and niches that the platform's original developers could not have possibly contemplated, much less had time to accommodate.

We have a long and proud history of this concept and this definition in the computer industry stretching all the way back to the 1950's and the original mainframe operating systems, continuing through the personal computer age and now into the Internet era. In the computer industry, this concept of platform is completely settled and widely embraced, and still holds going forward.

The key term in the definition of platform is "programmed". **If you can program it, then it's a platform. If you can't, then it's not.**

So, if you're thinking about computing on the Internet, whenever anyone uses the word "platform", ask: *"Can it be programmed?"* Specifically, with software code provided by the user? If not, it's not a platform, and you can safely ignore whoever's talking -- which means you can safely ignore 80%+ of the people in the world today who are using the term "platform" and don't know what it means. (Yes, there are hardware platforms too! But those are different and I'm not talking about them.)

Now, traditionally in the field of computing, there has been a single main way of providing a platform. You provided a computer system -- a mainframe, a PC operating system, a database, or even an ERP system or a game -- that contained a programming environment that let people create and run code, plus an API that let them hook into the core system in various ways and do things.

To quote **Tony Stark**, "That's how Dad did it, that's how America does it, and it's worked out pretty well so far."

The Internet -- as a massive distributed system of many millions of internetworked computers running many different kinds of software -- complicates things, and gives rise to **three new models of platform** that you see playing out in the Internet industry today.

I call these Internet platform models *"levels"*, because as you go from Level 1 to Level 2 to Level 3, as I will explain, each kind of platform is *harder to build*, but much *better for the developer*. Further, as I will also explain, each level typically supersets the levels below.

As I describe these three levels of Internet platform, I will walk through the pros and cons of each level as I see them. But let me say up front -- **they're all good**. In no way to I intend to cast aspersions on what anyone I discuss is doing. Having a platform is always better than not having a platform, period. Platforms are good, period.

So, let's walk through the three levels of Internet platforms.

Level 1 is what I call an "Access API".

This is the kind of Internet platform that is most common today. This is typically a platform provided in the form of a *web services API* -- which will typically be accessed using an access protocol such as REST or SOAP.

Architecturally, the key thing to understand about this kind of platform is that *the developer's application code lives outside the platform* -- the code executes somewhere else, on a server elsewhere on the Internet that is provided by the developer. The application calls the web services API over the Internet to access data and services provided by the platform -- by the core system -- and then the application does its thing, on its own. That's why I call this an "Access API" -- the key point is that the API is accessed from outside the core system.

This is of course the approach taken by eBay, Paypal, the Google Search API (before they killed it), Flickr, Delicious, etc. *Whenever someone pulls photos out of Flickr to put them on another web site, or whenever someone creates a Google Maps mashup, they're using a Level 1 Internet platform.*

This is undoubtedly a very useful thing and has now been proven effective on a widespread basis. However, the fact that this is also what most people think of when they think of "Internet platform" has been seriously confusing, as this is a *sharply limited approach* to the idea of providing a platform.

What's the problem? **The entire burden of building and running the application itself is left entirely to the developer.** The developer needs to provide her own runtime system, programming language, database, servers, storage, networking, bandwidth, and security, and needs to take responsibility for running all of the above -- and then exposing the application to users. This is a **very high bar** in terms of both **technical expertise** and **financial resources**.

As a consequence, you don't see that many applications get built relative to what you'd think would be possible with these APIs -- in fact, uptake of web services APIs has been nothing *close* to what you saw with previous widespread platforms such as Windows or the Mac.

This is, however, by far the easiest kind of Internet platform to create. As the platform owner, you don't have to worry about developer code running inside your system or developer functionality injecting itself into your system; you retain completely control over your user interface; and you can sharply limit the load impact that third parties can have on your systems -- i.e., throttling is straightforward.

Because of this and because Level 1 platforms are still highly useful, notwithstanding their limitations, *I believe we will see a lot more of them in the future* -- which is great. And in fact, as we will see, Level 2 and Level 3 platforms will typically all incorporate an Level 1-style access API as well.

Level 2 is what I call a "Plug-In API".

This is the kind of platform approach that historically has been used in end-user applications to let developers build new functions that can be injected, or "plug in", to the core system and its user interface.

For example, Photoshop has for a long time had a widely used and highly successful plug-in API. Lots of people have used the Photoshop plug-in API to build **tons of new functionality for Photoshop** ranging from support for new file formats to new ways to retouch images to new special effects to apply to images.

More recently, Firefox is well known for having a great plug-in, or extension, API that lets third parties build **a wide range of Firefox plug-ins**. These plug-ins span functions from blogging to dowloading to search to language translation.

In the Internet realm, the first Level 2 platform that I'm aware of is the **Facebook platform**.

When you develop a Facebook app, you are not developing an app that simply draws on data or services from Facebook, as you would with a Level 1 platform. Instead, you are building an app that acts like *a "plug-in" into Facebook* -- your app literally shows up within the Facebook user experience, often as a box in the middle of a page that Facebook otherwise defines, such as a user profile page.

Your Facebook app can of course *also* use Facebook's Level 1-style access API -- their web services API -- to pull data or services from Facebook's core systems -- and in fact the two approaches neatly complement each other, because without the access API your embedded Facebook app wouldn't know anything about the system in which it was embedded and wouldn't be very useful.

I think that Facebook's platform approach is a harbinger of a large number of new Internet plug-in APIs that will be created for lots of other Internet services from here on out. Which is great: developers will be able to *inject new functions into many other Internet services* in the future, just like they can with Facebook today.

As a historical side note, in retrospect, this is what AOL should have done in the mid 1990's when the web first popped up. At that point, AOL had a huge user base relative to the consumer Internet. However, AOL was completely closed -- third parties couldn't build new functions or apps that could plug into AOL and be used by AOL users. As a consequence, all of the creativity and third-party effort that AOL *might* have harnessed had they provided a plug-in API -- a way for third parties to build apps that would inject new functions into the AOL user experience -- went to the web instead. A few years later, it became clear to AOL users that the web is where all the interesting stuff was, and then when broadband came along and people had to switch ISPs anyway, the users bailed on AOL.

Seen through this lens, Facebook -- and I mean this in the best possible way -- is the new AOL, but, by also being a platform, executing the opportunity correctly.

Technically, with an Internet plug-in API approach such as Facebook's, the third-party app itself lives outside the platform -- outside the core system -- just like I described for Level 1 platforms. *The code for the app runs somewhere else.* This means that just like with Level 1 platforms, **the entire burden of building and running a Level 2 platform-based app is left entirely to the developer** -- who still needs to provide her own runtime system, programming language, database, servers, storage, networking, bandwidth, and security, and who still needs to take responsibility for running all of the above.

As a result, *the technical expertise and financial resources required* of a Level 2 platform's developer -- if she intends to build a meaningful app -- are *very high*.

Technical expertise: the developer has to be a world-class expert at building and deploying Internet applications, which have lots of moving parts.

Financial resources: the developer has to foot the bill for servers, storage, networking gear, bandwidth, etc., which can be significant for meaningful apps -- *especially if they succeed*.

In fact, *if an app succeeds on a Level 2 platform*, **the technical and financial burdens on the developer can rapidly become overwhelming** -- leading me to summarize with the phrase "success kills".

On top of that, there's an issue for the platform provider as well. When a third party app embedded into a Level 2 platform is down, because the developer doesn't know how to scale or doesn't have the money required to do so, the error shows up as an error *within the core system*. For example, whenever an individual Facebook app is down, users see the error within their Facebook pages -- even though Facebook itself is not responsible for the app outage, nor could Facebook do anything about that outage even if they wanted to. Ordinary users will not realize who is at fault and will tend to blame the core system -- in this case, Facebook.

For these reasons, I think that while Level 2 platforms are clearly very powerful and are wonderful for users, these issues *sharply constrain the number of developers who can build apps* to those who have the technical capability and the financial resources to build their own primary Internet systems anyway -- which is a *small fraction of the people who will ultimately want to be developers* on popular Internet platforms.

The great news, though, is that unlike a Level 1 platform where the burden of exposing the app to users is also placed on the developer, Level 2 Internet platforms -- as demonstrated by Facebook -- will be able to *directly help their developers get users for their apps*. This is one of the reasons I have called the Facebook platform a breakthrough -- Facebook provides a whole series of mechanisms by which Facebook users are exposed to third-party apps automatically, just by using Facebook. This is *great* for developers, and hopefully new Level 2 Internet platforms will follow in Facebook's footsteps -- and not in MySpace's, which could have been a Level 2 Internet platform well before Facebook but instead took a very hostile stance towards third-party developers.

It is also worth nothing that *Level 2 platforms are significantly harder to create than Level 1 platforms*. Facebook, for example, had to anticipate and provide technical solutions for a whole series of issues -- user interface issues, security issues, performance issues, caching issues, etc. -- to provide their plug-in API that providers of access APIs don't have to worry about. This is perhaps why we haven't seen more Level 2 platforms -- yet.

Level 3 is what I call a "Runtime Environment".

In a Level 3 platform, **the huge difference is that the third-party application code actually runs inside the platform** -- developer code is uploaded and runs online, inside the core system. For this reason, in casual conversation I refer to Level 3 platforms as **"online platforms"**. Let me explain.

- **A Level 1 platform's apps run elsewhere, and call into the platform via a web services API to draw on data and services** -- this is how Flickr does it.
- **A Level 2 platform's apps run elsewhere, but inject functionality into the platform via a plug-in API** -- this is how Facebook does it. Most likely, a Level 2 platform's apps *also* call into the platform via a web services API to draw on data and services.
- **A Level 3 platform's apps run inside the platform itself** -- the platform provides the "runtime environment" within which the app's code runs.

In addition, it is highly likely that a Level 3 platform will also superset Level 2 and Level 1 -- i.e., a Level 3 platform will typically also have some kind of plug-in API and some kind of access API.

Put in plain English? A Level 3 platform's developers **upload their code** into the platform itself, which is **where that code runs**. As a developer on a Level 3 platform, you don't need your own servers, your own storage, your own database, your own bandwidth, nothing... in fact, often, all you will really need is a browser. **The platform itself handles everything required to run your application on your behalf.**

Obviously this is a huge difference from Level 2. And this difference -- and what makes it possible -- is why I think Level 3 platforms are the future.

Let's start with one big issue right up front: Level 3 platforms are *much* harder to build than Level 2 platforms.

As a platform provider, once you accept the idea that user code -- code that you didn't write and you can't vet for quality or security -- is going to run *within* your platform, you have a whole pile of issues you have to deal with that a Level 2 platform can simply ignore.

The Level 2 platform's apps will be running their code elsewhere -- at the end of the day, the running code is someone else's problem. With a Level 3 platform, all the running code for all the apps is *your* problem.

What are some of those issues? To list a few: You have to provide a *runtime environment* that can execute arbitrary third-party application code. You have to build a *system for accepting and managing that code*. You have to build integrated *development tools* into your interface to let people develop that code. You have to provide an integrated *database environment* suitable for applications to store and process their data. You have to deal with *security* in many different ways to prevent applications from stepping on one another or on your system -- for example, sandboxing. You have to anticipate the consequences an application succeeding and needing to be automatically *scaled*. And you have to build an automated system underneath all that to provide the *servers, storage, and networking capabilities* required to actually run all of the third-party applications.

And you probably also have to provide Level 2 functionality -- a plug-in API -- and Level 1 functionality -- an access API -- so that third-party applications can actually do useful things once they are running within your system.

The bad news is that this is a truly intense technical and business undertaking, and not for the faint of heart.

The good news is that what it makes possible is magical.

Here's what's magical: the level of technical expertise required of someone to develop on your platform drops by at least 90%, and the level of money they need drops to $0. *Which opens up development to a universe of people for whom developing on a Level 2 or Level 1 platform is prohibitively difficult or expensive.*

Actually, it gets *even better than that*. You can provide an open source ecosystem *within* your platform to let users freely share code with one another -- actual running code! You can in essence have your own open source dynamic within your platform -- in the best case, allowing users to clone and modify one another's applications with a level of ease that the software industry has never seen. The rate of rapid evolutionary application development that can result from this approach will, I think, be mind-boggling as it plays out.

You can *also*, if you want, provide a marketplace that lets people buy and sell code -- then you can have the open source dynamic *and* the profit incentive. The sky's the limit in terms of how much development can happen on a platform like that.

The Level 3 Internet platform approach is ironically much more like the computer industry's typical platform model than Levels 2 or 1.

Back to basics: with a traditional platform, you take a computer, say a PC, with an operating system like Windows. You create an application. The application code runs right there, *on the computer*. It doesn't run elsewhere -- off the platform somewhere -- it just runs right there -- technically, within a runtime environment provided by the platform. For example, an application written in C# runs within Microsoft's Common Language Runtime, which is part of Windows, which is running on your computer.

I say this is ironic because I'm not entirely sure where the idea came from that an application built to run on an Internet platform would logically run *off* the platform, as with Level 1 (Flickr-style) or Level 2 (Facebook-style) Internet platforms. That is, I'm not sure why people haven't been building Level 3 Internet platforms all along -- apart from the technological complexity involved.

However, I think this will change, because *the advantages of being a Level 3 platform -- particularly the advantages to the developer, and thereby for the platform -- are so overwhelming*.

So who's building Level 3 Internet platforms now?

First, I am -- Ning has been built from the start to be a Level 3 platform.

I'll be writing more about this in another post, but in a nutshell, Ning is a full online platform for creating and running social networking applications. We provide all of the platform functions I described above, including the ability for users to either create their own applications or run clones or modified copies of applications we or other people provide.

There are close to 100,000 such applications currently running on the Ning platform -- you can see them at **Ning** in the form of all of the various social networks and applications in the system -- and that number is growing very quickly.

Second, in a completely different domain, Salesforce.com is also taking a Level 3 platform approach -- Salesforce now provides quite sophisticated ways for users and developers to create and upload code and program the Salesforce platform from a browser.

Salesforce provides a Level 3 platform both because it lets users easily customize Salesforce to do whatever they need, and also because it definitively trumps the criticism they historically got from packaged software vendors like Siebel who accused Salesforce of not being as adaptable as a piece of software you install on your own servers.

You probably don't see this in action much -- unless you're a Salesforce user -- but they're doing really interesting work in this area and getting great results.

Third, and again in a completely different domain, Second Life is a Level 3 platform.

It's a little different in that Second Life provides its own client -- for its immersive 3D world -- but you can think of the Second Life client as analogous to a browser in that it's an Internet client that draws on content being sent down from Second Life's servers.

Then, within Second Life, there is a complete runtime environment for running code provided by any user -- you can create items within Second Life and program them to do anything and behave in any way that you can possibly imagine. And you can, with permission, look at the code of another user's item and then make a copy or modify it to do whatever you want. And all of that code is running on Second Life's servers.

It's a tremendously dynamic platform that lets users build a dizzying array of worlds and objects that are all completely customized -- programmed.

Fourth, Amazon is -- I would say -- "sort of" building a Level 3 Internet platform with EC2 and S3. I say "sort of" because EC2 is more focused on providing a generic runtime environment for any kind of code than it is for building any specific kind of application -- and because of that, there are no real APIs in EC2 that you wouldn't just have on your own PC or server.

By this, I mean: Ning within our platform provides a whole suite of APIs for *easily building social networking applications*; Salesforce within its platform provides a whole suite of APIs for *easily building enterprise applications*; Second Life within its platform provides a whole suite of APIs for *easy building objects that live and interact within Second Life*. EC2, at least for now, has no such ambitions, and is content to be more of a generic hosting environment.

However, add S3 and some of Amazon's other web services efforts to the mix, and you clearly have at least the foundation of a Level 3 Internet platform.

Interestingly, Amazon's FPS -- Flexible Payments Service -- is itself a Level 3 Internet platform. You actually upload code written in a specialized programming language -- which they called GK, for

"Gatekeeper code" -- that controls how payments happen; that code runs within Amazon's online systems. It's a really innovative way to provide a highly flexible vertical service -- and great to see!

Fifth and last, Akamai, coming from a completely different angle, is tackling a lot of the technical requirements of a Level 3 Internet platform in their "EdgeComputing" service -- which lets their customers upload Java code into Akamai's systems. The Java code then runs on the "edge" of the network on Akamai's servers, and is distributed, managed, and secured so that it runs at scale and without stepping on other customers' applications.

This is not a full Level 3 Internet platform, nor do I think Akamai would argue that it is, but there are significant similarities in the technical challenges, and it's certainly worth watching what they do with their approach over time.

These examples illustrate one final point about Level 3 platforms: **you have to commit to never killing your platform**. This is a sharp difference.

Think about it: For a Level 1 or Level 2 platform, **if you kill the platform, you still have a working and useful system**. If the Google Search API gets killed -- and it was! -- *you still have Google Search* which is still useful to its users. If the Facebook platform gets killed -- which presumably it won't be -- *you still have Facebook* which is still useful to users as a social networking service in exactly the same way it was before they introduced their platform.

On the other hand, **if you kill a Level 3 platform, you destroy the whole reason people use your system to begin with** -- to develop and run custom code. If you remove the platform from Ning, Ning is useless -- applications don't run, and users can't do anything. If you remove the platform from Salesforce, all the users who are using customized apps can't use Salesforce anymore. If you remove the platform from Second Life, none of the objects or experiences in the virtual world work anymore and the whole user experience collapses.

Like my old boss Jim Barksdale used to say, it's the difference between the chicken and the pig at a ham and egg breakfast. The chicken's involved but the pig is committed.

I believe that in the long run, all credible large-scale Internet companies will provide Level 3 platforms. Those that don't won't be competitive with those that do, because those that do will give their users the ability to *so easily customize and program* as to unleash supernovas of creativity.

I think there will also be a generational shift here. Level 3 platforms are *"develop in the browser"* -- or, more properly, *"develop in the cloud"*. Just like Internet applications are "run in the browser" -- or, more properly, "run in the cloud". The cloud being large-scale Internet services run on behalf of users by large Internet companies and other entities. I think that kids coming out of college over the next several years are going to wonder why anyone ever built apps for anything other than "the cloud" -- the Internet -- and, ultimately, why they did so with anything other than the kinds of Level 3 platforms that we as an industry are going to build over the next several years -- *just like they already wonder why anyone runs any software that you can't get to through a browser*. Granted, I'm overstating the point but I'm doing so for clarity, and I'm quite confident the point will hold.

Three closing notes!

First, how will we see the new platforms of the future?

We are used to seeing platforms ship *as products* -- you buy and install a PC or a server and you build an app that runs on it, or equivalently you download and install an open source platform such as Perl or Ruby and you build an app that runs on it.

The platforms of the future won't be like that. **The platforms of the future will be online services that you will tap into over the Internet**, perhaps with nothing more running locally than a browser. They won't have anything you download, or even an SDK. They will look more like services than software. To paraphrase the Book of Matthew, "you will know them by their URLs".

Second, beware overfocusing on the apps of the past when thinking about the platforms of the future.

Lots of people got confused by the idea of apps running in the browser because when they thought of apps, they thought of the apps they used already on their PCs -- Word, Excel, Powerpoint -- and not the apps that would get built on the web -- eBay, Amazon, Salesforce.com. Now, it turns out in the fullness of time that word processing, spreadsheets, and presentation apps are also moving into the web -- as **Google is demonstrating**. But way before *that* happened, the web led people to create lots of *new* kinds of applications that were *not* possible on the PC.

A new platform typically enables a new set of applications that were not previously possible. Why else would there be a need for a new platform?

But: keep this in mind; *look for the new applications that a new platform makes possible*, as opposed to evaluating the new platform on the basis of whether or not you see older classes of applications show up on it right away.

Third, lots and lots of people have opinions about platforms, but the people whose opinions *matter* are programmers, and people who can make decisions about what programmers program.

This sounds incredibly elitist, to which I say: first, *the whole point of platforms is that they let people program on them*. So people who aren't programming, or making decisions about programmers programming on them, don't have a lot to do with them, and often don't know what they're talking about. Second, is is *really easy to learn how to program* -- in fact, it's never been easier. So there's really no excuse for anyone who wants to have opinions to not learn to program -- in fact, that's a great excuse *to* learn how to program!

And on that note, I'm going to go to sleep now and dream about "for" loops and "if/then" statements.