Responsive Delivery

A methodology for mobile development and deployment

by Ike DeLorenzo

Moovweb
San Francisco, California

July 2013
rev. 1.1

ABSTRACT

Responsive Delivery is a methodology employed to deliver HTML5 user experiences on a variety of devices—typically mobile and tablet—by programmatically transforming the user experience available on an existing, traditional web site. The transformed interface (often delivered to mobile, tablet, or kiosk devices) interacts with the same back-end infrastructure as the source web interface. The result is to achieve advantages in ROI, time-to-market, maintenance-cost, and end-user experience quality over competing methodologies.

The goal of Responsive Delivery is to preserve business logic, workflow, and assets as each transformed user experience targets a new device. In this way, Responsive Delivery seeks to obviate the need for the backend engineering which would otherwise be required to support additional devices and user experiences.

Responsive Delivery (RD) is often seen as a broader, business-oriented successor to RESS (Responsive Web Design with Server Side Components). RESS itself arose out of attempts to address the limitations of strict RWD (Responsive Web Design). Each methodology is widely used, and each works best in different situations. The evolution suggests a shift from RWD's initial all-on-the-client approach to the more practical, balanced separation of server and client-side logic used in RESS and —in a more structured way—in RD.

> The goal of Responsive Delivery is to preserve business logic, work flow, and assets as each transformed user experience targets a new device.

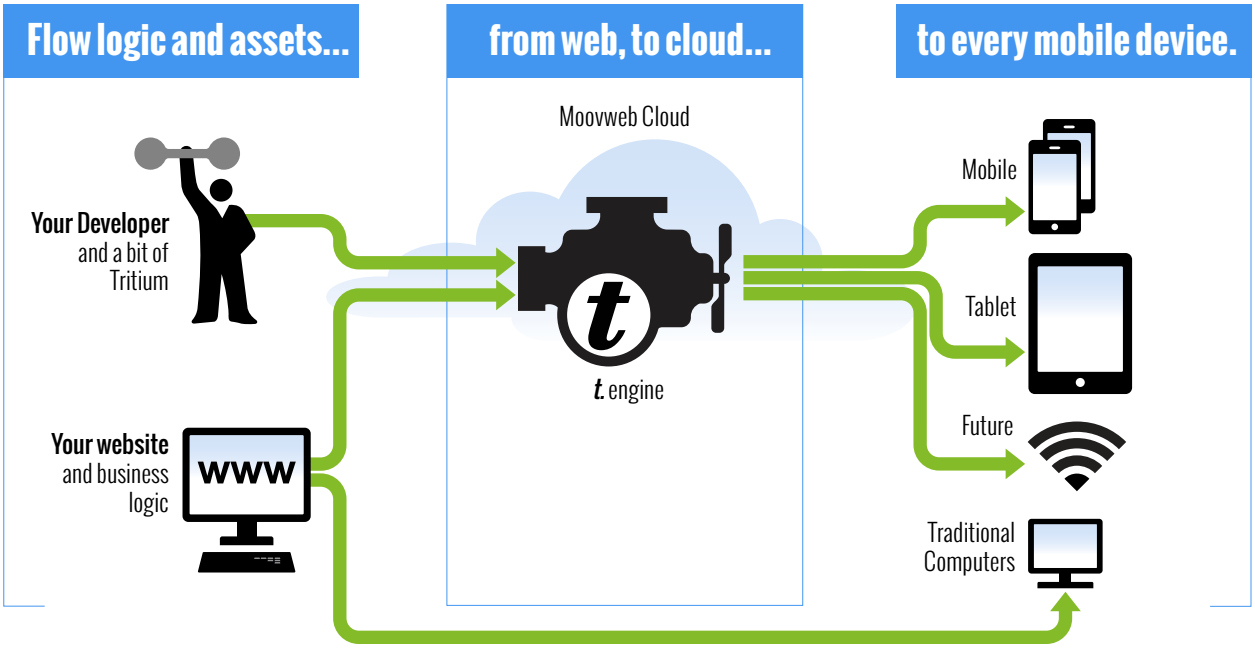


figure 1.1 Overall system diagram of Moovweb's cloud-based Responsive Delivery platform

Moovweb, based in San Francisco, California, is the leading cloud-based provider of Responsive Delivery, via its Moovweb platform.

Using a mobile device strategy including Responsive Delivery, businesses can leverage the web site(s) they already have to power a variety of new mobile experiences using only front-end developers and well-known technologies like CSS and HTML.

"RESPONSIVE" AND THE EVOLUTION OF MOBILE DEVELOPMENT

Mobile devices—as we consider them today—emerged with the introduction of the iPhone (2007), Android for multiple devices (2009), and the iPad (2010). By 2010, development of new mobile experiences for a range of screen sizes became necessary for a variety of businesses, led by the retail sector.

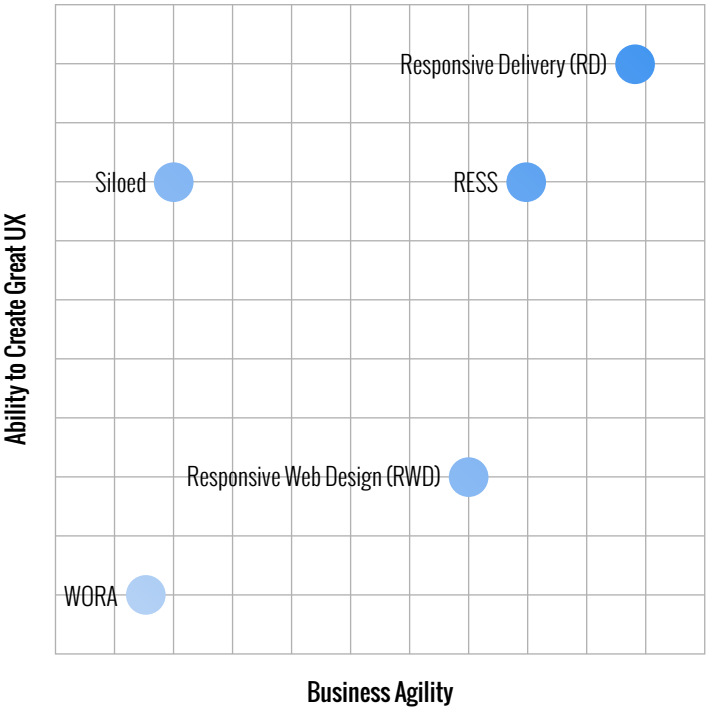
MOBILE 1.0: SILOS AND THE LEGACY OF WAP

In the beginning, before 2009, mobile development took a siloed approach where custom front-end code was developed for each targeted device, and custom back-end code was developed to drive each experience. In addition, mobile web sites and mobile apps were usually independent and unrelated efforts. This approach was acceptable largely because of the small universe of mobile devices that existed. Such a siloed approach had been the norm for mobile development up to that time for early "feature phone" applications over a protocol called WAP (Wireless Access Protocol).

But siloed development did not work well for the more complex mobile sites that were arriving with ever-more full-featured browsers: the iPhone and, later, Android. Multiplying front-end work (and backend work) to deliver siloed mobile sites and apps was becoming untenable for businesses in an environment where the number of devices on the market was poised to rise, and applications strived to be more complex.

MOBILE 2.0: RESPONSIVE WEB DESIGN (RWD)

In 2010, a methodology for accommodating the rising number of devices (and other issues) was introduced in the web development magazine "A List Apart." The idea was fully explained in 2011 in the 150-page book titled "Responsive Web Design." RWD, as the methodology is now known, was simple to explain and had immediate appeal: Use



the features of CSS, JavaScript, and HTML5 to create a single web site that auto-sizes itself to the width of the target device. (It had recently become possible to determine the width of the client device screen in major browsers).

Few other methodologies existed at the time to address user interface development across desktop and portable devices. As mobile boomed, and these devices proliferated, the profile of RWD rose.

The early adopters of RWD were small top-tier web shops of designer-developers who closely followed such thought-leading blogs. The elegance and puzzle-like challenge of RWD appealed to them: Create a single codebase that will unpack itself in the client browser and render appropriately. It was that same quest that had fueled client-side Java and all its preceding write-once-run-anywhere "WORAs." The idea that RWD might be a way to achieve this long-sought Holy WORA Grail proved irresistible, and RWD chatter on the blogs spread the buzz.

When RWD was implemented by small groups of very talented programmers, with large budgets, and ample time, the sites were quite good: fast, interesting, beautiful, and nicely optimized for each screen size. Several high-profile RWD sites were produced this way, which encouraged wider adoption. Design shops that were able to master the methodology took on a rock-star status in the web design community.

But in real-world business conditions (limited budgets, limited time, existing infrastructure) in-house developers had trouble building effective user experiences with RWD. The marquee results that had been so blogged-about proved difficult to achieve in real-world business conditions with real-world programmers.

RWD works best when there are a few screen sizes (called "breakpoints" in RWD) to consider. The introduction of odd screen sizes in Samsung's Galaxy devices (2011), and the introduction of the iPad mini (2012) complicated RWD implementations, as code complexity will increase markedly with additional device breakpoints.

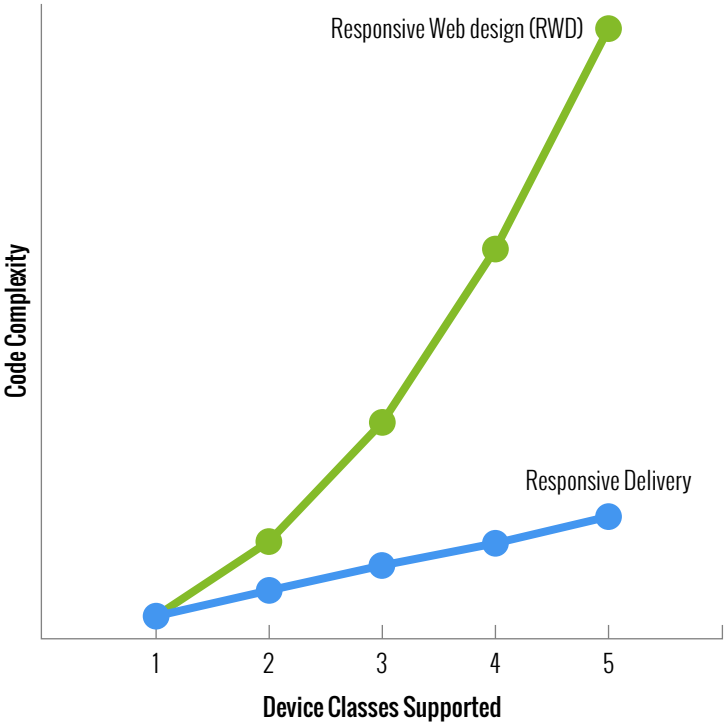


fig1.3 Code complexity increases exponentially as device breakpoints are added to RWD client code (e.g. mobile phone, phablet, iPad Mini full size tablet, desktop). When devices are coded as separate endpoints, as in Responsive Delivery, code complexity increases in a linear relationship to the supported endpoints.

MOBILE 3.0: RESS and RESPONSIVE DELIVERY

By 2012, the pressure on strict RWD led to the introduction of a more moderate methodology called RESS (Responsive Web Design with Server Side Components). RESS reduces the amount of code sent to the mobile device by sending only that code necessary for that specific device, or for a group of close-in-size devices, such as similar-sized mobile phones.

In this way RESS reduces and simplifies the HTML, CSS, and JavaScript that is sent to a given device, There are no RWD-style "breakpoints" in the client code, which greatly increases maintainability after initial deployment. Within a limited range of screen sizes, the user interface adjusts and presents properly.

RESS detects screen size on the server, usually reading the browser ID string, a technique known as user-agent sniffing. RESS then uses server-side languages such as PHP (to send various subsets of HTML, CSS, JS) and Sass (to generate the CSS for the target device) to create a small, device-specific codebase for each targeted device or group of devices.

The advantage of RESS is to reduce code complexity for both development and maintenance, and to allow greater customization of user interface on each of the range of devices targeted. RESS, properly implemented, is a good Goldilocks solution between absolute-client and absolute-server for mobile UI.

RESPONSIVE DELIVERY: BEYOND UI, TO UX AND BUSINESS LOGIC

Methodologies that help create UI are only part of a multi-device deployment solution. Mobile sites, of course, also need workflows to do something useful. Responsive Delivery arose as a way to flow business logic and workflows to multiple devices, along with the assets and UI.

> The methodology of Responsive Delivery is a way to refactor existing business logic and assets via a define-then-automate process known as *transformation*.

Coding business logic for mobile deployment had been a separate backend development process in RESS, RWD, and siloed development. In most of these cases, the required code and logic had already been implemented for the traditional desktop web site. Enterprises needed to develop separate, parallel backend code to drive their mobile sites, and/or create and publish various types of APIs. This kind of duplicate backend development—or re-development—typically consumes about 80% of mobile development effort.

The methodology of Responsive Delivery is a way to refactor existing business logic and assets via a define-then-automate process known as "transformation." The existing backend infrastructure is not touched. The result in mobile web development is to effectively import the user experience (UX) of the existing desktop site, then transform it—usually into streamlined HTML5—for mobile delivery. Such transformations are live, so source site content updates flow through to the various mobile "endpoints" (phones, tablets, kiosks).

RESPONSIVE DELIVERY IN PRACTICE

Moovweb, a San-Francisco based enterprise software company, provides a cloud-based Responsive Delivery platform that is widely used to mobilize large corporate websites. Source web site transformations are defined in standard front-end languages.

The company's flagship platform is comprised of three parts. A set of developer tools allows front-end programmers to fashion mobile experiences from imported business logic, workflows, and assets of the existing web site—known as the "source site" in RD terminology. Coding requires only front-end languages, any or all of: HTML, CSS, JavaScript, and the Tritium scripting language.

Domain-specific languages (DSLs) like Tritium are a common way to facilitate tasks that would be complex to accomplish in general languages such as JavaScript. As with most DSLs, Tritium is designed to be easy to learn—it is syntactically similar to JQuery—and powerful for a very specific task, here isolating the DOM objects and workflows of the source site and transforming them. Tritium script was developed at Moovweb under the direction of Sass inventor Hampton Catlin.

Moovweb's cloud-based *"Control Center"* provides user management, code packaging (into *"projects"*), and push-to-live management for launching the projects to the Moovweb cloud.

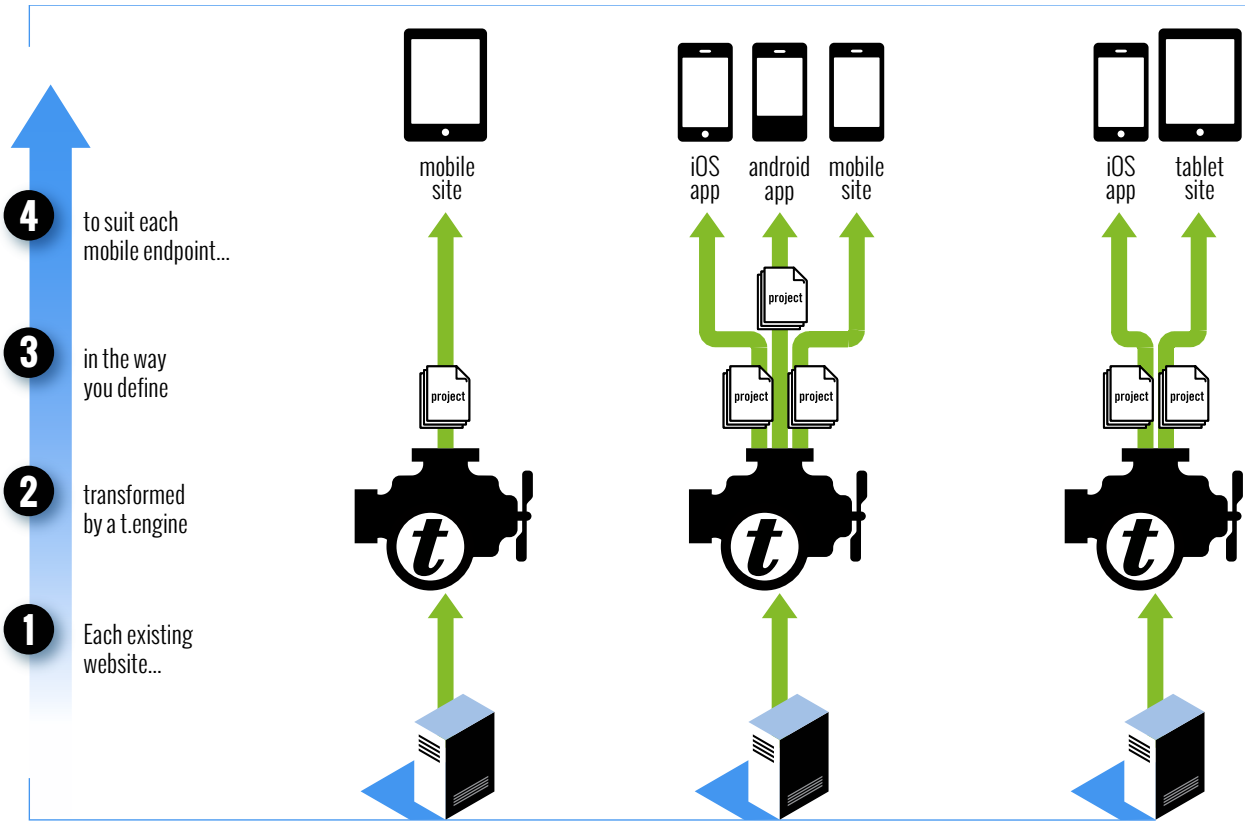


figure 1.2 Enterprise example: One company with three subdomains. Responsive Delivery via Moovweb here uses three t.engines (one for each domain), and one, two, or three projects per engine to target mobile endpoints.

One transformation engine, or "*t*.engine", is provisioned in the Moovweb cloud for each source site an enterprise intends to transform and mobilize. A single *t*.engine can mobilize a single source site to many endpoints: smartphone browser, tablet, iOS app, etcetera. For each endpoint (separate device or app), a project is coded to define the transformation desired for that endpoint. Projects, written largely in Tritium, "run" on the *t*.engine. In this way a single t.engine provides an easy to maintain collection of projects that are expandable—enterprise programmers can just add another project to target another endpoint.

These *projects* are the equivalent of RWD breakpoints, or conditionals in RESS. They define the right code destined to run on each device. RD projects eliminate the tangled interrelatedness and complexity of RWD code, and have the RESS advantage of running on the server. They also provide the unique business advantage of providing access to existing back-end business logic that is absent in previous responsive methodologies.

BUSINESS ADVANTAGES OF RESPONSIVE DELIVERY

TTM, TCO

Of the major responsive methodologies, RD offers the best time-to-market, and the lowest TCO for mobile sites. RD's time-to-market is also the fastest for hybrid mobile apps, and RD's ongoing app TCO is roughly equivalent to RESS. (RWD is seldom used to create hybrid mobile apps.)

Additionally, RD can be used to furnish business logic and assets to native apps. This is not possible with older responsive methodologies like RWD and RESS.

The average buildout time, from planning to launch, for an enterprise mobile site using RD is about three months. Contrast that with RWD, where high-profile projects take over a year, and small enterprise projects take about nine months. RESS projects fall at about 6 months, still nearly double the time-to-market of RD.

Significantly, second and third RWD projects undertaken by major corporations have shown little economy of scale or progressive improvement in time-to-market. Data here is sparse as corporations often switch to RESS if the first attempts at pure RWD fail (RESS projects do shown good economy of scale as more projects are undertaken).

MANAGEABLE, MAINTAINABLE CODE STRUCTURES

RWD advocates often point to the "single code base" or "single file" that defines (and indeed executes) on each client browser as an advantage in code development and maintenance. In reality, RWD does require separate code for each device or group of devices. In RWD these separate code sections are "breakpoints" in one large code file that must be sent, in its entirety, to every targeted device.

In RD the same separations exist, but are intentionally separate files packaged as "projects." Enterprises want separate projects per endpoint so they can have different engineers on each project. (Enterprise-level mobile sites addressing b2c, b2b, and b2e use cases can be complex with many stakeholders.) RD projects are more maintainable, less tangled, and less complex than a single "ball" of code.

And, of course, multiple programmers can work on multiple files much more effectively than when all contending for the same one dependent codebase. As in all software development, code can be shared across

projects if desired—but now sharing is voluntary, and recommended for common code.

With the small, self-described "rock-star" RWD teams, the engineers can share one ball of code just fine—often they are sitting in the same room all day. This informal and expertise-dependent situation is not viable as a distributed, enterprise programming environment.

And so, RWD has been successful for small teams, and at small web design agencies. But the methodology has difficulty scaling as you add more than 5 or 6 engineers. The clean endpoint separation available in RD (one project per endpoint) helps engineers work in parallel, greatly reducing development time and, when well-coordinated, increasing code reliability.

Such separation is available, but not required. A traditional RWD "one-ball" approach is fully supported by Moovweb's platform for teams and sites that want to take existing web logic and assets and deliver them to mobile endpoints in a traditional RWD code structure for selective rendering on the client.

SEO

Recent changes in the way Google indexes mobile sites gives some advantage to Responsive Delivery, due to the reduced code volume of RD and its flexibility of delivering either one URL, or several separate URLs per device (at the discretion of the enterprise).

In the past Google had favored sites that essentially duplicated all content across traditional web (desktop), tablet, and mobile. Now a set of search related tags are supported by Google allowing site authors to indicate how the sites relate, and how the company wants them to be indexed. In the case of Moovweb's RD solution, these tags are supported and management is largely automatic.

The search engine no longer biases indexing on whether a single URL (www.company.com) or several URLs (m.company.com, t.company.com, touch.company.com) are used. In this way companies using RD can choose which URL strategy meets their marketing objectives without risk of SEO issues. There is also evidence that the faster page load and rendering times typical of RD improve page rankings on Google.

SECURITY

Real world business web sites, and the infrastructures that drive them, have been built out gradually over the past decade or more. They are

> Responsive Delivery is the most secure mobilization methodology because it keeps the web security mechanisms in place, without opening new doors (APIs, XML,etc.) to the backend infrastructure.

often large, complex, and have been created by different architects and engineers who may or may not still be at the company. As anyone who has worked in IT in such an enterprise knows, documentation and coherence are often lacking.

But the web layer of these infrastructures does effectively expose the business logic and assets that form the user experience and relationship between the company and its customers or clients. Put simply, the workflows and experiences a company needs to provide to users are available on its desktop web site.

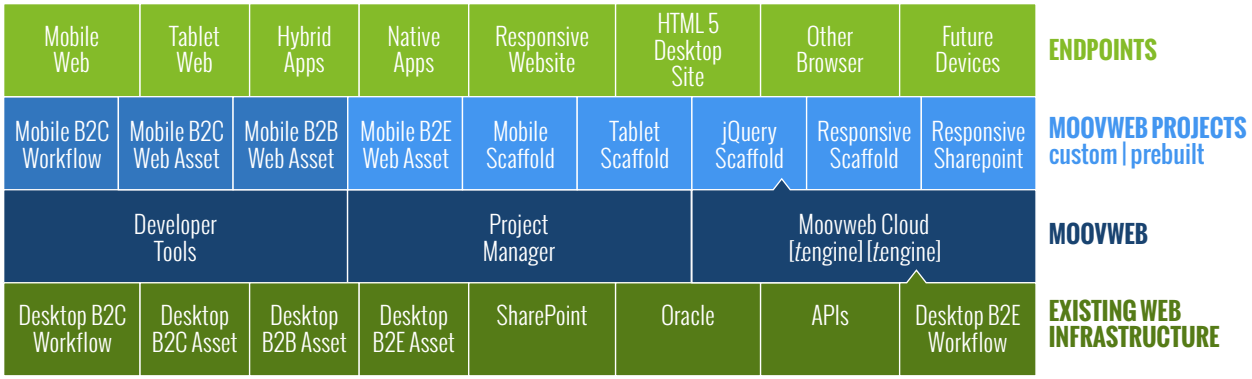
Importantly, the security of a mature web layer has been addressed and empirically proven over time. Firewalls, application server security mechanisms, database call sanitization, and a layers of other mechanisms protect the web layer from intrusion attempts that have been methodically addressed by IT—from SQL injection, to buffer overflow attacks, to input validation exploits.

Responsive Delivery is the most secure mobilization methodology because it keeps these empirically proven web security mechanisms in place, without opening new doors (APIs, XML,etc.) to the backend infrastructure. Since no back-end coding is done at all, no security flaws will be introduced. In multi-component infrastructures, it is precisely these openings that create security risk. As internet security firm Net Square noted at this year's BlackHat conference: "Why fight the wall when you've got an open door?" The wall they refer to is the existing, protected web site.

TOWARD A "UX-INFORMED INFRASTRUCTURE"

From an architectural standpoint, Responsive Delivery is essentially a layer on top of an existing multi-tier IT infrastructure. This layer enables the existing architecture to function as the base of a "UX-informed infrastructure": one designed to deliver UX to multiple heterogenous devices.

Existing web IT infrastructures—typically three layers—have been built with [1] the data tier as a foundation layer, [2] business logic (or "process logic") as the middle layer, and [3] presentation (web) logic as the top and outward-facing tier. The presentation layer has historically been charged with serving user interfaces that communicated the middle business logic layer, which was itself fueled by the lowest data tier. In theory, the presentation layer is to be architecturally



independent of business logic and data, acting cleanly and independently as a presentation mechanism.

In practice, the presentation layer—especially in long-existing systems—has business logic intertwined with the user interface logic. The tiers cannot easily be separated, or used independently. And, in many cases, the presentation layer often side-steps the process logic layer entirely and (for expedience) communicates directly with the data layer (or database). The reason for this is years of implementation pressure. When user interfaces are interactive, and have logic that is dependant on previous user actions, real-world programmers tend to abandon the theoretical elegance of layer separation in favor of getting the user experience coded, done, and launched. Moovweb forms a UX transformation layer on top of the existing IT infrastructure which is —in reality—acting as a single monolithic source of business logic and data, with the web-facing component as its most stable and accessible interface.

User experiences, especially on the web, need to react quickly to human users. So business logic and data are regularly stored and manipulated in the presentation layer for efficiency and speed (of both execution and engineering). Think session cookies, "cached" data, customized CSS, the various scopes of PHP and Apache variables, the data associated with security and personalization. Often these kinds of data, are moved all the way up the client—say, a web browser—where end users interact with logic and data that may or may not make its way back to the lower IT layers soon, or ever.

As anyone who has worked in real-world enterprise IT can tell you, the so-called "three tier architecture" of theory usually does function as a single-tier entity in practice: interdependent, tangled, built by a decade of engineers in ways that have not been perfectly coordinated. Despite this, we do have today web user experiences that work well, that consistently bring end users through the paces of required business

logic, and that deliver data to enterprise-critical databases. The goal here is to add new devices, and new user experiences to these existing infrastructures, without the cost and risk of major IT surgery.

MOBILE SITES AND APPS: EQUAL CITIZENS

IN THE BEGINNING

In the Summer of 2008, Apple introduced the "app" to the mobile world —and anyone with a television—with the launch of its iPhone "App Store." At this time, and for the next two years, all App Store apps were required to be written in native code (Objective C) using Apple's SDK. Apple took (and continues to take) a 30% cut of each app's list price. Within a year, there were 50,000 third-party apps available for the iPhone, up from the 12 from that Apple itself shipped pre-installed on the device.

Growth of the App Store quickly began to be strangled by lack of Objective C programmers, a hard language to learn and not typically a language mastered by front-end developers and user interface designers. By late 2009, the Apple Store had competition from other App Stores serving other mobile platforms: Blackberry World, Google Play, Nokia Store, Samsung Apps Store, and the inevitability of a Windows Phone Store. Coding the same app for two or more platforms required mastering two or more languages and SDKs—and then actually writing the identical (one hoped) app from scratch.

NATIVE HITS ITS BUSINESS LIMITS

Then, as now, the best user experiences were being written by front-end programmers using HTML, CSS, and JavaScript. These front-end developers were coding mobile apps as what were essentially one-page HTML "apps" using a variety of wrapper-frameworks that allowed the HTML app to run in a web view within a native app shell. This technology bridged the gap between HTML and native code. The most popular of these frameworks, now owned by Adobe and widely used today, was PhoneGap.

As PhoneGap stabilized, in late 2009, Apple began allowing PhoneGap apps—essentially HTML5 apps—to be sold in its App Store. These apps become known as hybrid apps. The floodgates were open to

front-end developers, whose hybrid apps now could target multiple app stores with a single code base written in a familiar, well-supported stack (HTML5). Hardware functionality such as GPS, accelerometer, and camera eventually were accessible across these platforms by one JavaScript call made from HTML5. Today, a majority of new apps are hybrid, created by PhoneGap and other gap-closing app frameworks.

"RESPONSIVE" APPS?

It was during this same time period that Responsive Web Design was gaining a foothold in mobile web site development. As we have discussed, the core tenet of RWD is to determine the screen size, then have conditional HTML, CSS, and JavaScript adjust themselves while rendering in the browsers. For mature browsers, this worked well enough. But for the nascent web views of PhoneGap the quantity and conditionality of RWD code caused manifest rendering and execution problems (to some degree, mobile app web views have not kept pace with the mobile browsers on those same devices, and at this high level of code complexity they still do present execution problems).

More importantly, programmers who are looking for a way to write less code with less complexity (hybrid apps) did not want to write more code with more complexity (RWD). For this reason, companies who use RWD to deploy mobile sites are seldom able to leverage the code for hybrid app development efforts.

RESPONSIVE DELIVERY FOR APPS

Mobile has arrived at a point where many businesses want apps that bring consumer-quality UX as well as business/process logic to mobile users. (A departure influenced by the initial leisure-driven app market). In many cases, these business apps are deployed in addition to mobile and tablet sites; and some offer added functionality linked to GPS coordinates or other device functions.

Responsive Delivery is well suited to such a mobile strategy. The same cloud-based mobilization technology (for Moovweb, *t.*engines) that is used delivered to mobile sites can also deliver the lean HTML/CSS/JS that renders and functions best in an app's web view. Importantly, the cloud-based transformation code used to transform the source site is largely or wholly reused across RD "endpoints", e.g. mobile browser, Android app, iPhone app, iPad mini. In this way Responsive Delivery provides a unified solution across sites (mobile and otherwise), apps (phone and tablet), and fringe devices (internet TV devices, kiosks).

CONCLUSION

Design, development, and deployment of mobile experiences has evolved as fast as the devices themselves, both born just five years ago. Methodologies to meet user demand for UX and applications have evolved in the shortest time cycles in computing history—a dramatically compressed version of what we saw first in desktop software, and faster than we previously believed to be the most compressed lifecycles possible, the desktop web.

Mobile methodology, namely Responsive Delivery, has now arrived at the place each of these previous revolutions reached to achieve lasting productivity: a way to rapidly develop and deploy user experiences (browser, and apps of all sorts) with vetted, tested business logic and assets.

RD achieves its speed in development and deployment by leveraging the finished work of the previous two revolutions. It leverages backend IT built to fuel the web, and it leverages the existing web layer itself. In this way, RD empowers companies to focus human and financial capital solely on addressing those new mobile components that comprise the the current multi-device revolution: mobile UX, and the users themselves.