DRAFT

Emerging Web Development Paradigms and the Zeppelin Framework

Peter Elger
Richard Rodger
Eamonn de Leastar

Contents

Date

Scope

This white paper presents a case that the software development community, particularly the community developing cloud-based services and applications, is on the crest of another wave of innovation. The current software development stack, and associated platforms, may have run its course, and evidence is mounting that its replacement by a new wave of technology, architectural approaches in immanent.

Context

Evolution

The evolution of the Web has been well charted in recent years. With the coining of the term Web 2.0 (2004) a useful starting point, there is a reasonably comprehensive understanding of how the web as a platform has progressed since. This has included the stabilization of web services protocols, the rise of User Generated Content, the proliferation of Mobile web, the advent of the Smart Phone/Device/Tablet and the App Store Model.

However, the underlying tools, architectures and development practices have also been through an overhaul over this period and have made this innovation cycle possible. In particular, there has been a marked shift from the traditional "enterprise" stack (EJB2, .NET), and high ceremony development methods (RUP) to a more agile approach (XP), with a full embrace of more open tools and frameworks. Sometimes termed "lightweight" approach, this shift has included rapid evolution in web frameworks (Spring, Osgi) and the arrival of highly productive variations on these, most notably Ruby on Rails and its derivatives, usually bound to a relational database (MySql).

Coupled with this evolving stack, web browser performance, stability and capability has improved dramatically. The client side web is now clearly formed by the nexus of HTML, CSS and Javascript, with the latter in particular the lynchpin of significant innovation in the usability, power and flexibility of web applications. A set of

Javascript libraries (jQuery and others) has radically altered the usage patterns within the browser, unleashing unsuspected features in a robust environment. All three are sometimes grouped under the term HTML5, which in addition includes standardized approached to geolocation, 2D and 3D graphics, offline services, standard communications channels and more. Although not quite accurate, the term HTML5 usefully encapsulates the reach and ambition of the latest wave of browsers, and would seem to have significant momentum from all major players, hardware and software and infrastructure.

Revolution

There are signs, however, that this evolutionary approach may have run its course. The "lightweight" development stack of Relational Database, Component Service/Framework + Templating Engine, all running on a linux back end (a variant of the so called LAMP stack), is encountering a major shift in the underlying infrastructure - the arrival of the cloud. In particular, cloud based services coupled with advances in virtualization, have altered the principles around which applications have been architected to date. When this is also combined with various models for smart phone/tablet development, there is an argument that we are entering into another inflection point, comparable to the one foreseen in 2004.

What this particular movement will lead to is as yet unclear. However, it seems certain to yield new opportunities in services, mobility, flexibility and productivity in application development and deployment. In this context, there are signs of disruption within the current development stack. Although significant stability has been achieved since 2004, many its tenets are now being called into question:

- Database: The dominance of the relational database is not longer given. The "NoSql" movement is gathering pace with many open implementations of this broader, and perhaps more scalable architecture for the data store (MongoDB, CouchDB). When coupled with Googles Map/Reduce, it may be possible for more highly capable and intelligent systems can be constructed at a fraction of the cost for traditional relational systems.

- Middleware: Having already preceded though a series of major shifts over the past decade (rise and fall of Object Request Brokers, rise and fall of EJB, rise and stabilization of web frameworks), middleware is a useful touchstone when assessing the state of software and services. Evidence is mounting that the sheer complexity of current enterprise stack (J2EE, .NET) is causing profound limitations in the scale and reach of applications thus architected. The lightweight stack, evolved in some sense as an alternative to the traditional stack, may have reached its peak in Heroku, a marriage of cloud based services with a stable web framework. However, more disruptive technology is already emerging. In particular, the key to truly scalable services has always been the approach to concurrency. A radical alternative to traditional threading model (embodied in Heroku) is emerging. In particular, successive attempts to solve the concurrency problem (discussed below) are converging towards a more radical approach; namely the so-called non-blocking option.

- Client: The rise of the app store model is still taking shape. In particular, the introduction of this model to the general web (Google Chrome Web Store) may generate unforeseen consequences and trajectories in services and apps. For instance, the Chrome web store contains many applications that are indistinguishable from their apple app store equivalents (e.g. New York Times). However, these applications are full HTML5 (not native), are by definition more cloud oriented, and are thus liberated from highly restrictive (and complex) native app development toolkits.

The Rise of Javascript

A Prediction

The mainstream programming language for the next ten years will be JavaScript. Once considered a toy language useful only for checking form fields on web pages, JavaScript will dominate enterprise software development. Why this language and why now? Today, the Java language is the one to beat. Java dominates enterprise software development. JavaScript and Java may have similar names, but the similarity ends there. Though sometimes confused, they are very different languages. JavaScript owes its name to an accident of history; a failed and very strange marketing ploy from the early days of the web, when Netscape tried to leverage the growing popularity of Sun Microsystem's new Java language. Both companies have now retired from the industry.

JavaScript is the language that web designers use to build web pages. However, it is not (yet) the language the software engineers use to build the business logic for those same web sites. JavaScript is small, runs on the client, the web browser. It's easy to write unmaintainable spaghetti code in JavaScript. And yet, for all these flaws, JavaScript is the world's most misunderstood language. Douglas Crockford, a senior engineer at Yahoo, is almost singlehanded responsible to rehabilitating the language. In a few short, seminal online essay published shortly after the turn of the century, Crockford explains that JavaScript is really LISP, the language of artificial intelligence. JavaScript borrows heavily from LISP, and is not really object-oriented at all. This curious design was well suited to a simple implementation running in a web browser. As an unintended consequence, these same mutations make JavaScript the perfect language for building cloud computing services.

Here is a further prediction: within ten years, every major cloud service will be implemented in JavaScript, even those from Microsoft. JavaScript will be the essential item in every senior software engineer's skill set. Not only will it be the premier language for corporate systems, JavaScript will also dominate mobile devices. Not just phones, but also tablets. All the while, JavaScript will continue to be the one and only language for developing complex interactive websites, completing drowning out old stalwarts such as flash, even for games. For the first time in a history, a truly homogeneous programming language infrastructure will develop, with the same toolkits and libraries used from the top to the bottom of the technology stack. JavaScript everywhere

The Evidence

How can such a prediction be made? How can one make it so confidently? Because it has all happened before, and it will happen again. Right now, we are at a technology inflection point, yet another paradigm shift is upon us, and the JavaScript wave is starting to break. This is a familiar pattern; every ten years or so the programming world is shaken by a new language, and the vast majority of developers, and the corporations they work for, move en mass to the new playground.

Two technology shifts have preceded this one, and help to illustrate the shift. Prior to Java, the C++ language was dominant in the final decade of the last century. What drove the adoption of C++? What drove the subsequent adoption of Java? And what is driving the current adoption of JavaScript? In each case, cultural, technological and conceptual movements coalesced into a tipping point that caused a sudden and very fast historical change. Such tipping points are difficult to predict. No such prediction is made here – the shift to JavaScript is not to come, it has already begun. These tipping points are driven by the chaotic feedback channels at the heart any emerging technology..

What drove C++? It was the emergence of the object-oriented programming paradigm, the emergence of the PC and Microsoft

Windows, and support from academic institutions. With hindsight such large-scale trends are easy to identify. The same can be done for Java. In this case, the idea of the software virtual machine, the introduction of garbage collection – a language feature lacking in C++ that offers far higher programmer productivity, and first wave of internet mania. Java, backed by Sun Microsystems, became the language of the internet, and many large corporate networked systems today run on Java. Microsoft can be included in the "Java" wave, in the sense the Microsoft's proprietary competitive offering, C#, is really Java with the bad bits taken out.

Despite the easily recognizable nature of these two prior waves, one feature that both share that neither wave led to a true monoculture. The C++ wave was splintered by operating systems, the Java wave by competing virtual languages such as C#. Nonetheless, the key drivers, the key elements of each paradigm shift, created a decade-long island of stability in the technology storm.

The Nexus: Cloud, Mobile & HTML5

Cloud computing is one of the key drivers compelling the current wave of innovation. For the first time, corporations are moving their sensitive data and operations outside of the building. They are placing mission critical systems into the "cloud". Cloud computing is now an abused term. It means everything and nothing. But one thing that it does mean is that computing capacity is now metered by usage. Technology challenges are now solved by sinking capital into big iron servers. Instead, the operating expense dominates, driving the need for highly efficient solutions. The momentum for green energy only exacerbates this trend. Needless to say, Java/C# are not up to the job. We shall see shortly that JavaScript is uniquely placed to benefit from the move to cloud computing.

Mobile computing represents the other side of the coin. The increasing capabilities of mobile devices drive a virtuous circle of cloud-based support services leading to better devices that access more of the cloud, leading to ever more cloud services. The problem with mobile devices is the severe fragmentation. Many different platforms, technologies and form factors vie for dominance, without a clear leader in all categories. The cost of supporting more than one or two platforms is prohibitive. And yet there is a quick and easy solution: the new HTML5 standard for websites. This standard offers a range of new features such as offline apps and video and audio capabilities that give mobile websites almost the same abilities as native device applications. As HTML5 adoption grows, more and more mobile application will be developed using HTML5, and of course, made interactive using JavaScript, the language of websites.

While it is clear that the ubiquity of HTML5 will drive JavaScript on the client, it is less clear why JavaScript will also be driven by the emergence of cloud computing. To see this, we have to understand something of the way in which network services are built, and the challenges that the cloud brings to traditional approaches.

The Challenge of Concurrency

Threads

Diverse approaches to programmatically "coping" with concurrency have long been a source of contention among software developers. The evolution of the various approaches to concurrency is well illustrated in the C like languages, particularly Java. Although Java was designed with thread based concurrency in mind (unlike C & C++), its currency support has evolved significantly since its inception, with adjustments made to the core syntax, the libraries and the recommended approaches. The fundamental mechanism (synchronized keyword to serialize method access), has been supplemented with concurrent data structures, more expressive annotations, and an extensive rework of the concurrency model in Java 5 to incorporate an new "executor" framework. However, concurrent programming in Java is still regarded as a complex and error prone, with non-determinism an ever present worry, even for systems long deployed in the field.

Actors

The java concurrency module is founded on the shared state semantics of a single multi-threaded process, whereby threads can share resources and memory, but with locks associated with specific data structures. Alternatives to this model have gained some ground. The actors model rules out any shared data structures (and their resource hungry locks), with concurrency achieved by message passing between autonomous threads - each thread (an actor) has exclusive access to its own data structures. In functional languages derived from Java (Scala, clojure), immutability itself is elevated to be the default programming model. This requires wholesale adoption of functional approaches (or object-functions hybrids in the case of Scala), with the consequent profound change in programming style and heritage. With all of these approaches there is one common characteristic. Separate threads are created, with their own stacks and program counters. Although the opportunities for inter-thread synchronization vary, such synchronization must occur at some stage, with consequent overhead associated with task switching, memory usage and general processor load.

Non-Blocking I/O

There is an alternative, which has its origins in an era that predates the general acceptance of multi threaded infrastructure. Evolved to meet the requirements for responsive I/O in single processor systems, it sometimes takes the the term “Non Blocking IO”, although this term has also been applied to threaded designs. Originally devised as a set if interrupts and associated daisy chained interrupt handlers, in the modern sense (if we can call it that), non-blocking I/O implies and extensive use of callbacks in API design and usage. In this context, all opportunities for blocking are replaced by passing a callback parameter, to be invoked on completion of the deferred task or I/O request. A somewhat counter-intuitive programming style, it has been criticized for its verbosity and general awkwardness.

In certain programming languages it is indeed verbose - Java in particular is encumbered with a high-ceremony anonymous inner class syntax which make callbacks quite difficult to orchestrate. Also, in Java and other languages of that generation, the callbacks are limited in scope and place severe restrictions around the context they can access. What they lack is a “closure” capability - essentially a form of delegate/callback/function handle - which also carries (encloses) a well defined context that can be safely accessed when it is activated. Closures have become a hot topic in programming language recently, and Java itself is slated to this capability in future versions. JVM derived languages such as Scala and Groovy have this capability, as does Clojure via its Lisp heritage. In fact the term closure originates from these functional languages.

C10K Problem

This challenge is made concrete by what is known as the C10K problem, first posed by Dan Kegel in 2003. The C10K is this: how can you service 10000 concurrent clients on one machine. The idea is that you have 10000 web browsers, or 10000 mobile phones all asking the same single machine to provide a bank balance or process an e-commerce transaction. That's quite a heavy load. Java solves this by using threads, which are way to simulate parallel processing on a single physical machine. Threads have been the workhorse of high capacity web servers for the last ten years, and a technique known as “thread pooling” is considered to be industry best practice. But threads are not suitable for high capacity servers. Each thread consumes memory and processing power, and there's only so much of that to go round. Further threads introduce complex programming programs, including a particularly nasty one known as “deadlock”. Deadlock happens when two threads wait for each other. They are both jammed and cannot move forward, like Dr. Seuss's South-going Zax and North-going

Zax. When this happens, the client is caught in the middle and waits, forever. The website, or cloud service, is effectively down.

Event Based Programming

There is a solution to the this problem – event-based programming. Unlike threads, events are light-weight constructs. Instead of assigning resources in advance, the system triggers code to execute only when there is data available. This is much more efficient. It is a different style of programming, one that has not been quite as fashionable as threads. The event-based approach is well suited to the cost structure of cloud computing – it is resource efficient, and enables one to build C10K-capable systems on cheap commodity hardware.

Threads also lead to a style of programming that is known as synchronous blocking code. For example, when a thread has to get data from a database, it hangs around (blocks) waiting for the data to be returned. If multiple database queries have to run to build a web page (to get the user's cart, and then the product details, and finally the current special offers), then these have to happen one after other, in other words in a synchronous fashion. You can see that this leads to a lot of threads alive at the same time in one machine, which eventually runs out of resources.

The event based model is different. In this case, the code does not wait for the database. Instead it asks to be notified when the database responds, hence it is known as non-blocking code. Further, multiple activities do not need to wait on each other, so the code can be asynchronous, and not one step after another (synchronous). This leads to highly efficient code that can meet the C10K challenge.

JavaScript is uniquely suited to event-based programming because it was designed to handle events. Originally these events were mouse clicks, but now they can be database results. There is no difference at an architectural level inside the “event loop”, the place where events are doled out. As a result of its early design choices to solves a seemingly unrelated problem, JavaScript as a language turns out to be perfectly designed for building efficient cloud services.

Node.js

The one missing piece of the JavaScript puzzle is a high performance implementation. Java overcame it's early sloth, and was progressively optimized by Sun. JavaScript needed a serious corporate sponsor to really get the final raw performance boost that it needed. Google has stepped out. Google needed fast JavaScript so that its services like Gmail and Google Calendar would work well and be fast for end-users. To do this, Google developed the V8 JavaScript engine, which compiles JavaScript into highly optimized machine code on the fly. Google open-source the V8 engine, and it was adapted by the open source community for cloud computing. The cloud computing version of V8 is known as Node.js, a high performance JavaScript environment for servers.

All the pieces are now in place. The industry momentum from cloud and mobile computing. The conceptual movement towards event-based systems, and the cultural movement towards accepting JavaScript as a serious language. All these drive towards a tipping point that has begun to accelerate: JavaScript is the language of the next wave

Development Frameworks

It is only a few short years since Ruby on Rails (RoR) was the new kid on the block. At the time if its inception, RoR was a highly innovative development framework. The key driver for mass adoption of RoR was hugely increased developer productivity through "convention over configuration", an approach which has now certainly entered the zeitgeist and which has been adopted by almost all of the current development stacks: for example Python's Django or PHP's Cake.

The predominant application deployment model for most organizations during the rise of RoR was owned server infrastructure: i.e. make some capital investment in server hardware on which to deploy applications. Under this model operational expenditure was relatively static and was based on monthly costs for colocation and bandwidth. Operational efficiency of deployed, in the field, applications was not so important for anyone but the really large sites, as long as the application could scale horizontally to some degree, capacity could be added by purchasing more hardware which was a one off hit if it could be accommodated into existing cabinets / racks.

With the mass adoption of cloud computing, this model is flipped on its head. Deploying to the cloud requires little or no capital investment, however, operational expenditure is now directly tied to the efficiency of deployed applications. There is now a clear economic driver for efficient web applications and services.

Whilst advances have been made by the major languages and frameworks, fundamentally, they do not make the best use of the available compute resources and are therefore not best suited to operation in the cloud. Furthermore experience has shown that these frameworks suffer from a number of other deficiencies:

The SPA Disjoint
RoR type frameworks exhibit a Model View Controller (MVC) architectural structure. Under this paradigm, an application consists of a set of MVC triplets that are processed server side to render html back to the client. However, most modern applications no longer fit this model and are increasingly adopting the Single Page Application (SPA) or Multi-Single Page Application (MSPA) architectural style. Under this model static html is sent down to the client and acts as a basic application frame. Client side javascript then makes AJAX requests to "hang" the front end functional elements onto the application frame. Consequently much more of the application logic is implemented on the client in javascript. Whilst some work has been done in this area, notably backbone,js and Faux, none of the major MVC frameworks provide any governance or organizational structure for client side javascript. Developers are left to construct their own ad-hoc client side application architectures over libraries such as JQuery. This often means that the client can quickly degenerate to spaghetti code with little or no unit testing.

Code Duplication
The SPA Disjoint also leads to code duplication. Take for example the task of verifying and sanitizing user input. To provide a good user experience and rapid response time, this task is best done on the client. However for security reasons it must also be checked server side. Therefore this logic is typically implemented twice, once in javascript on the client and again on the server in the whatever language is appropriate to the framework being used (ruby, python, etc...).

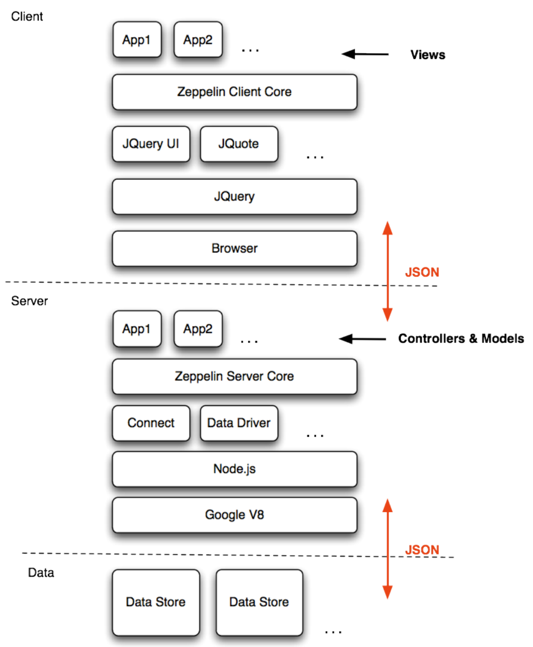
Relational Database Assumption
RoR type frameworks were built around the assumption that the framework would talk to a single relational database. Indeed in early versions of rails it was difficult to introduce an additional relational datastore into an application (1). All the current production frameworks make the unstated assumption that the back end is an SQL compliant database.

Language Proliferation
To work end to end with any current MVC framework one must be proficient in a minimum of five languages - SQL, one of Ruby/Python/Php..., Javascript, HTML and CSS. This can have one of two effects, either individual developers must context switch between the various languages depending on where they are in the stack at a given point in time, or a team is broken down into "front end" and "back end" specialists, a division which can cause delay and communication overhead when implementing application functionality.

Project Zeppelin

Zeppelin is a low friction, all javascript, distributed MVC development framework that supports multiple applications per server stack instance. Zeppelin addresses many of the issues outlined above, whilst at the same time preserving and indeed, we believe, improving upon all that is good about the current set of MVC application stacks, most notably developer productivity.

Zeppelin is built on top of Node.js, connect, JQuery and JQuote. A schematic of the framework architecture is depicted in the figure below.



Distributed MVC

Zeppelin implements a distributed MVC architecture, wherein view processing is offloaded to the client tier. View templates implement the familiar <%= %> syntax however the template language is javascript. Zeppelin is designed to work best with NoSQL type data stores such as mongodb and implements a plugin type architecture for datasource drivers. Drivers must implement a JSON only northbound interface into the framework.

All Javascript, Low friction.

Zeppelin is all javascript from the data store to the client. This design reduces the number of required languages for users of the framework from five (for RoR Django etc...) to just three, HTML for layout, CSS for design and javascript for code, irrespective of where the code is executing. This approach has some profound consequences for the application developer:

- The mental disjoint between client and server development is reduced, one can think in javascript all of the time. This is a deeper, more fundamental shift than using tools such as GWT.
- Code can be shared between the client and server, for example helper modules for tasks such as data input validation can be written once and executed on both tiers.
- Ensuring that the framework deals exclusively with JSON data means that one can again think in javascript when searching for and manipulating data on both the client and server tiers
- Use of NoSQL data sources / stores means that the frame work is schema-less, requiring no database migrations or other such constructs

All of which servers to reduce the mental friction required to use the framework and hence speed up developer productivity.

Cloud Framework

Zeppelin use of a non-blocking core platform, Node.js, provides for a highly efficient operational environment out of the box. Furthermore Zeppelin strives for operational scalability through two key architectural choices:

- View processing is handled on the client, meaning that less processing resource is required on the server
- Use of NoSQL data stores mean that there is no requirement for the framework to flatten data to JSON or other format before sending to the client. Indeed if the document format is clearly thought through the server component can be reduced to a simple conduit through which JSON data can flow to the client.

Multiple Applications per Stack
Zeppelin supports the deployment of many individual applications onto a single stack instance, providing an efficient mechanism for could based deployment and hosting.

Conclusion

Todo

1 http://magicmodels.rubyforge.org/magic_multi_connections/)