T E C H N I C A L P U R S U I T I N C .

WE WRITE JAVASCRIPT SO YOU DON'T HAVE TO.™

The Zen Of TIBET™

THE PROBLEMS, PRINCIPLES, AND PRIORITIES DRIVING TIBET'S DESIGN

Introduction

We got started with mission-critical web applications in 1997.

In 1997, years before we started creating what would eventually become TIBET, we built our first mission-critical web application. The goal was to port 100+ order-entry screens for 11 product lines from Delphi to the Web. Deployment targeted a nine-state region of 800+ sites. Response times were capped at 5 seconds. Applets and plugins were forbidden.

Our first web project pushed all application logic into the client...so we know what's possible.

It took us eight months and 50,000 lines of JavaScript to get our wizard-driven order-entry system ready for acceptance testing. By then, all presentation and session logic ran in the client. The client and server communicated via service calls and a precursor to JavaScript Object Notation (JSON). We called it Client/SOA — client/server for the Web.

We believe:

The virtual machine that won is the web browser.

Client/SOA will replace centralized web design.

JavaScript authoring can't scale enough.

Through our experience building similarly complex web applications over the subsequent years we came to believe a few key things:

- web browsers would become the dominant platform, making markup, CSS, and JavaScript critical languages for application development.
- a Client/SOA or "independent-client" architecture for web applications was inevitable. Server-centric would give way to client/server...again.
- JavaScript-centric authoring wouldn't scale to meet the demands imposed by the backlog of applications to be ported to the web.

TIBET reflects these beliefs as well as many others we hold.

Like most good products, TIBET reflects the philosophy of its creators. Over the years our beliefs about web architectures and authoring models have bubbled up, infusing the entire platform with a unique personality.

Our goal here is to relate the central philosophies behind TIBET. We'll be doing that by stepping you through the problems, principles, and priorities that have driven its construction and evolution to this point.

Our goal is to help you decide if TIBET is the right choice for you.

With a better understanding of the "Zen of TIBET" you'll be able to make a fully-informed decision about whether TIBET is a good fit for you, your teams, and your projects.

Let's start by taking a look at the problems TIBET was built to solve.

The Problems

The problems TIBET addresses are implicit in two key beliefs we hold regarding today's web architectures and web authoring models[1].

Web architecture and authoring model are key.

We feel a) migration to an offline-ready, independent-client architecture is inevitable and b) JavaScript-centric authoring cannot scale to meet the demand of porting millions of desktop applications to the web:

Independent Client: A client that works when the user needs it to.

- Current libraries are too small and require too much JavaScript skill,
- There simply isn't a scalable pool of JavaScript developers available,
- Our development processes waste the time of those who do exist.

There are multiple issues centered around current JavaScript approaches.

Too Small A Library

Ironically, we never thought the problem with JavaScript frameworks was that they were too big. If anything they were, and still are, far too small.

Ironically, most libraries are actually too small.

Your application can't run offline, i.e. it can't support an independent and fully-mobile architecture, unless you port it to the client. You can't do that without a library powerful enough to replace what's on the server.

You can't run offline unless you port to the client.

Try to imagine loading the standard library from Ruby, Java, or Python into the web browser. Imagine doing it efficiently. That's the mission.

That takes a full-scale, fast-loading library.

Too Few Developers

To be honest, we didn't think the world needed another JavaScript framework in 1999 and it doesn't need another one now -- at least not one that's going to simply trade one set of JavaScript APIs for another.

Yet Another JavaScript API isn't likely to solve the underlying issues.

Today's billions of web pages weren't built by programmers; they were built by people using markup and CSS. If we're going to scale web development we need to find a way to return to authoring in markup and to make that markup smarter, more extensible, and more shareable.

The web didn't reach its scale by writing code. It was built using markup.

---

[1] http://technicalpursuit.com/docs/TheAxesOfTIBET.pdf

The backlog of web apps left to build is immense.

Too few JS developers exist to get it all done.

In the same way there were, and remain, an almost unlimited number of web pages to create, there were, and remain, an almost unlimited number of web applications to port from VB, Delphi, PowerBuilder etc.

There simply aren't enough JavaScript programmers to get it all done, regardless of how large or powerful a JavaScript API we give them.

Too Many Reloads / Time Sinks

Reload cycles waste an immeasurable amount of development time.

Every time you hit reload you've wasted the most precious and limited development resource you have -- time. You've also incrementally disrupted your development flow, negatively impacting productivity.

There are a number of variations on this theme of wasted time and disrupted flow but we see hitting Reload as the most consistent signal available that as an industry our development process needs work.

Every core technology we use is dynamic.

Every core technology of the web is dynamic from HTML to CSS to JavaScript and yet our development processes ignore that reality.

Our development cycles don't leverage the power inherent in dynamism.

Instead of directly manipulating our applications as they run we engage in the development equivalent of banging rocks together to make fire by relying on arcane edit, reload, login, navigate, type, debug cycles. Some teams even add a compile step or two into their process. It's crazy. Our web development productivity is suffering death by a thousand reloads.

Solve These Equations Simultaneously

Solutions to individual issues exist, but can make other issues worse.

You could try solving problem #1 by creating a framework with hundreds of classes and thousands of methods. Early versions of TIBET did just that. But if you don't couple that approach with a solution to problem #2 you've simply built a massive framework for a non-existent workforce.

You could try solving problem #1 by using other languages and generating code, but in that case you're making both #2 and #3 worse by filtering an already limited talent pool and adding compilation delays.

A true solution to these issues must solve them simultaneously.

We believe these problems have to be solved in a unified and coherent fashion if we're going to move the needle on serious web development.

Design Beliefs

The solutions we've come up with are the result of a number of pretty opinionated design beliefs. Some of these will cause strong reactions. You don't have to agree with them. Just be aware that if you choose TIBET you're going to encounter the impact of these beliefs quite a bit.

Some of our beliefs may trigger strong reactions. We're OK with that.

Web Applications Run In The Client, Not The Server

We believe the focal point of a web application, or any application for that matter, is the end user. As a result, we believe the client, not the server, is where the application runs. The question is how efficiently.

Web applications run where the user is, not where the server is.

When the user starts their browser they have a goal in mind and if they could achieve that goal without waiting on your server they would. Your server might provide access to valuable data but it's also in their way, adding latency with every DNS lookup, HTTP request, and database call.

If the user could run the app without your server they would.

From the user's perspective every server call is a remote procedure call, something we're taught to avoid if we're concerned about performance. This perspective is borne out by web performance data which shows HTTP overhead is by far the primary determinant of web performance.

The user's HTTP calls are essentially RPC calls.

If the data were on the user's hard drive they'd gladly run without those server calls. In other words, users would run offline if they could, and sync to the server as needed. Users want an independent client, one that works when they need it to, not when they can get a connection.

Running offline and syncing as needed would serve users the best.

Once we realized the real implications of running offline we started building TIBET that way, offline, without a server, from the file system.

TIBET is built offline so we know the benefits.

Admittedly, our approach is 180 degrees off from how most developers work with the web. Then again, we've never lost time waiting for the server team to get something running so we could work on our apps.

We don't wait on server teams or server delays.

The sense of freedom we experience by developing offline, never waiting on a server, is the experience we want to provide to end users of TIBET.

We want users to have that experience.

Focus On Using Markup, CSS, and JavaScript. In That Order

Browsers understand HTML, XML, CSS, JavaScript, and JSON.

For better or worse browsers understand markup (HTML/XML), CSS, and JavaScript/JSON. As mentioned earlier, you can use other languages to generate code but the less your team works with native technologies the more you cripple their ability to debug and maintain things over time.

We focused on markup over JavaScript and arrived at "smart tags".

With TIBET we focused on markup, CSS, and JavaScript and went to work embracing, encapsulating, and extending. By prioritizing on markup first, then CSS, then JavaScript we eventually arrived at "smart tags".

Imagine your app as a single, sharable, tag.

Imagine a tag, the <corp:travel/> tag. That's your travel application. That tag expands into header, content, footer. Those expand into other tags and so on until you reach HTML. That's the model we focused on.

TIBET is semantic tags all the way down.

With TIBET it's tags all the way down. TIBET puts shareable, smart, semantic markup at the center of your web application authoring.

Agility Requires Strength And Flexibility

Agile is great, if you have the right foundation.

There's a lot of talk in our industry about agile development, one month delivery cycles, and the like. They're a great idea -- provided you've got Smalltalk, Ruby, or the JDK behind you so you've got something to refactor into while you work at the speed of thought. JavaScript though?

You need more than JS provides, or most light frameworks offer though.

How agile can you be with JavaScript's 8 types and ~150 functions? Or another hundred thanks to a lightweight framework? How fast can you be when there's more functionality missing than there is in your toolbox?

We think apps should be 80% markup and 80% library. Not 80% custom.

We think applications should be 80% library code and 20% your code, not the other way around. With our markup-first focus we also think web apps should be 80% markup and 20% code, not the other way around.

We use Smalltalk as our model environment.

We don't think it's an accident Extreme Programming, the precursor to agile, arose within Smalltalk's rich environment. As a result we look at Smalltalk for cues to the kind of environment we need for JavaScript.

Everything Interesting Happens When You Least Expect It

The three things a web application needs to communicate with most: the user, the window, and the server, are going to respond asynchronously.

The user, window, and server are all async.

Most modern JavaScript frameworks respond to the demands of async development through heavy use of anonymous callback functions.

Callbacks are one way to manage this reality.

The problem with callback-style development is it doesn't scale. After a certain size, callback-centric applications turn to functional spaghetti. In addition, callback-style development is notorious for poor error handling.

Callback-based code doesn't scale well. It turns to spaghetti.

A little redirection solves these issues. TIBET uses generic callbacks and error traps to translate low-level events and errors into TIBET Signals.

Signals give us a way to respond to everything from interface events, to server push notifications, to raise() calls in our own code, using a simple and consistent approach that doesn't inevitably lead to callback hell.

We use a centralized event hub to organize things so they scale.

There's Probably Already A Standard For That

As the saying goes there's nothing new under the Sun.

When it comes to working in the software industry you can be fairly sure that if you go back far enough you'll find that the problem you think is totally new and unique and challenging was solved by someone in 1968.

Very little in development is actually new.

Our approach with TIBET has been to ignore that...until we've produced an initial design we think meets our criteria. Once we've done our design brainstorming without skewing the results by looking at existing solutions we step back and look for any solutions or standards that we could potentially integrate so that we're not re-inventing the wheel.

We do our design without skewing it from history, but then we look for any standards we can apply.

As a result TIBET includes support for a host of W3C, IETF, Oasis, and LISA standards including varying levels of support for: XMPP, XForms, XML Events, XML Schema, XML Base, XLink, XInclude, XPointer, XSLT, XPath, and TMX. And those are just the XML standards.

As a result we support dozens of key internet standards, particularly for enterprise XML.

Messaging, Encapsulation, and Extremely Late Binding

TIBET: Class-based OO.

TIBET is unabashedly Object-Oriented (OO); Class-based OO.

We use class-based OO to remain maintainable at extremely large scale.

Why class-based OO when modern JavaScript frameworks seem to be more functional? Because experience tells us above 30,000 lines or so functional JS is hard to maintain; and functional coders are hard to find.

Alan Kay, who defined OO, calls for 3 things.

Alan Kay, who defined the term Object-Oriented, calls out 3 essential OO elements: messaging, encapsulation, and extremely late binding[2].

Messaging

Messaging implies that objects serve to organize application functionality.

The implication behind messaging is that objects become the means by which application logic is organized. As a result we felt it was key to "do objects right", meaning we wanted objects without the limitations we saw in existing JavaScript frameworks, limits absent in Smalltalk.

TIBET ensures all forms of messaging work right.

We wanted types that could inherit from each other, instances that could rely on messaging their type, dependable call-super semantics, and the ability to refactor the hierarchy without having to patch hardcoded calls.

That means making OO work right as well.

For us, depending on messaging means being equally able to depend on an object foundation that reduces maintenance, not the inverse.

Encapsulation

Encapsulation isn't native to JS, but can be implemented w/closures.

Without some syntactic gymnastics JavaScript doesn't natively support encapsulation. You can work around this limitation with closures but only by sacrificing performance and testability. We don't think it's worth it.

We don't use that model. We rely on convention and tooling which don't force poor trade-offs.

In TIBET property prefixes of _, $, or $$ denote 'protected', 'private', or 'internal'. A pair of accessors, get() and set(), access properties via reflection so you never use a prefix directly. This latter convention means future tooling can confirm property access doesn't violate encapsulation by ensuring _foo, $foo, and $$foo are never used outside an accessor.

---

[2] http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

Extremely Late Binding

"Extreme late-binding of all things." That's Alan Kay's third requirement. JavaScript does that naturally at a low level but failure to focus on late-binding at a higher level of your API can lead to fragile, inflexible code.

"Extreme late-binding of all things." We took that to heart the most.

TIBET's accessors use high-level late binding to refine their actions. For example, get('name') will look for a getName method before accessing the 'name' property. Adding getName to existing code "just works".

TIBET's property accessors look for more refined calls at runtime.

TIBET makes extensive use of two patterns in addition to our accessor lookup approach which further demonstrate our focus on late binding in the API. We refer to them as Call Next Method and Call Best Method.

We also have other patterns of late binding we use extensively.

Call Next Method

Class-based OO languages often provide a mechanism by which you can invoke methods on an ancestor in the inheritance hierarchy. This call-super feature is powerful but most JS frameworks have you hard-code class references. The result is a fragile hierarchy and potentially incorrect call chains as a result of "early-binding" by the developer.

We don't hard-code type references to 'call super'.

TIBET implements a callNextMethod() method you call instead. TIBET's callNextMethod() logic is aware of the native JS call chain and uses it plus reflection to call the right method and ensure it's properly bound.

TIBET uses a form of late-binding to support a callNextMethod method.

Call Best Method

In TIBET there are a number of method pairs such as as() and from(), isa() and validate(), format() and parse(), which work with types using our call-best-method pattern, another form of high-level late binding.

There are a number of ways Types are leveraged as part of late-binding.

Imagine you have a object of type 'Foo' you want as a string. In TIBET you invoke object.as(String). The as() method uses reflection to find the best method. If Foo has an asString method that'll be called; if String has a fromFoo() method that's called. Inheritance is checked as well so if Foo inherits from Bar and String implements fromBar() that'll be called. TIBET's metadata makes reflection-driven API and late binding possible.

Our callBestMethod uses reflection, Types, and double-dispatch to find and invoke the best method at runtime.

The Magic's In The Meta. Metadata. Metatypes, Meta*

TIBET uses definition methods, not raw JS syntax, for metadata.

Unlike other JS frameworks, but in ways similar to Smalltalk and Ruby, TIBET uses methods to define types, properties, and most every other part of your application. And these "method methods"[3] track metadata.

TIBET uses a MetaType, not Function, for Types. Types inherit fully.

TIBET types aren't instances of Function, they're instances of MetaType, a special type which defines properties all TIBET types inherit. As it turns out most of what Types inherit from MetaType relates to code definition.

In TIBET you use methods such as defineSubtype, defineMethod and defineAttribute to define your application's components. These methods do several things but a key one is tracking runtime metadata.

Having metadata means TIBET can support much smarter authoring tools.

Having complete metadata on the types, methods, properties, constants, and other parts of your application means TIBET is not only able to do smart things at runtime for your code, it means we're able to build smart tools which can help you develop your application at runtime as well.

Suspend Your Disbelief And Set The Amp To 11

Moving the needle means questioning not only the answers but questions, suspending disbelief, going to 11.

If you're going to design something that truly moves the needle one of the first things you have to do is to question not only the conventional answers but the conventional questions. Suspend your disbelief. Stop telling yourself that's crazy it'll never work. Just dial your amp to 11, fire up the whiteboard, and see what happens.

Questioning the questions helps ensure you're solving the right set of problems.

For example, instead of trying to answer the conventional JS question of "How do we make the library smaller?" we questioned the question. Why is smaller relevant? Smaller loads faster. Ok. So the real issue is how to load things faster. We focused on faster, not smaller, and came up with an entirely different way to boot large web applications efficiently.

Don't be afraid to ignore conventional wisdom.

Never be afraid to ignore conventional wisdom, it might simply reflect a set of good answers to the wrong questions.

---

[3] Douglas Crockford, *JavaScript: The Good Parts,* (Yahoo! Inc, 2008)

Coding Principles

Coding style/standards are a common source of disagreement for many in the software community and we're no exception. We disagree with much of what we've seen adopted as best practice in JavaScript code. TIBET reflects those differences of opinion in sometimes dramatic fashion. Don't be surprised if you disagree with what follows. That's fine. You don't have to write TIBET source, you just get to leverage it :).

Coding standards are pretty contentious; these are ours.

Self-Documenting Code Is An Oxymoron. Document Intent

We don't believe in self-documenting code.

Self-documenting code doesn't exist, at least not in our experience.

Code is the implementation of your intent, the translation of your intent into a programming language. If your intent and your implementation vary but all you captured in the file is the implementation what then?

Comments are for intent. Code is implementation. They aren't redundant.

For our code we've found it useful to sketch out public APIs we envision in the form of rough documentation to get a feel for how developers will interact with the system. We know the documentation may change, but it helps ensure documentation is part of the deliverable from day one.

If you're doing TDD you need intent (comments) so you can write tests.

As you iteratively build out the actual implementation we believe inline comments describing your local intent, algorithms, assumptions, etc. are just as key to creating maintainable code for those who follow you.

Inline comments define your intent, not your implementation.

This document/deliver cycle can be as small and iterative as you like. One function, a broken test, a working test. A new method stub, a test, a working implementation. It's intention-driven implementation.

Yes, our approach can lead to code that may have what appear to be redundant comments. Good comments aren't redundant, they come first and express intent. The code simply expresses an implementation.

A file with nothing but comments is a spec.

A file with nothing but code is an executable.

Source files need both.

A file with nothing but comments is a spec. A file with nothing but code is an executable, and unmaintainable over time. A file with both contains a living spec bound to an implementation. Verify both. Maintain both.

Never Trust A Browser When Your Project Is On The Line

We've ported TIBET for over a decade to dozens of browser versions.

They all exhibit bugs sooner or later.

Our first JavaScript project of serious scale was in mid-1997. Back then it was IE3 and Nav3. Then it was IE4 and Nav4. Then Firefox, Opera, IE5, IE6, IE7, Safari, Chrome, IE8, IE9, IE10. Stable foundation? Not hardly.

Some versions of Firefox broke function definition. Some versions of Safari broke HTTP result codes. IE? Well, we know that story too well.

We learned two hard lessons from porting TIBET from browser to browser to browser since 1999:

You can't feature detect for the entire JS, HTML, XML, and CSS spec set.

#1 - You can't feature detect enough to ensure complete compliance to the JS specification any more than you can feature detect for layout issues in the UI. You have to use both user-agent and feature lookup.

You can't trust native calls to remain stable.

#2 - Wrap every native function your code relies on in an API you can use to patch underlying bugs when, not if, they arise. Because you can rest assured they WILL arise. Browsers are NOT a stable foundation.

Encapsulate everything.

As a result of these lessons TIBET encapsulates browser APIs as well.

Make It Work. Make It Right. Make It Fast.

Work. Right. Fast. In that order.

Our development mantra is "Make it work, make it right, make it fast."

Always bet on Moore's Law.

We may be old-school, but we believe that if you build something to work, then make it work right, you might find that Moore's law has taken care of the last one for you. If not, at least you end up tuning the right thing. Premature optimization is the root of all evil after all.

With TIBET we designed what we felt was the right markup, CSS, and OO infrastructure. Admittedly at first it was slow. But that was then.

JavaScript performance is 100x what it was a few short years ago. And getting faster every year.

Today's browsers run JavaScript orders of magnitude faster than even just a short year ago and improvements are coming faster every year.

Today TIBET is fast without sacrificing any of the power we intended.

Shipped Code Lives Forever; Get It Right, Right Now

Agility notwithstanding, bad code lives forever once it's been shipped.

Applications deploy longer than you think and what were supposed to be temporary hacks end up deployed into production. The theory is you'll refactor on the next iteration. The reality is management, marketing, or your users are demanding new features and you'll never get the time.

Frameworks are even more sensitive to this problem. A framework is, or should be, the concrete foundation your application builds upon. It should be solid, consistent, dependable. We take that goal seriously.

The public interfaces and APIs we offer in TIBET need to stand the test of time. We can't be constantly redesigning, subjecting your team to the fallout of a process consisting of "design by successive approximation".

In building TIBET we've tried hard to get it right, right now.

If you ship it you may as well have chiseled it on stone tablets. You will never rewrite that code.

Frameworks are even more sensitive since they are other people's foundational framing.

Get your API right the first time; there is no second time.

Craftsmen Treat Exceptions As First-Class Use Cases

Most JavaScript code is notable for its attention to nothing but the path a user takes through the code when everything goes according to plan.

We believe a framework can't afford to be cavalier about the use cases associated with what we like to call the real world, the world where your net connection dies, your file access fails, your user enters the wrong kind of data, or the library you included turns out to be buggy.

Exception handling is poorly named if for no other reason than it makes it seem as if exceptions are just that, exceptions. But they're not. They're equally valid and in some ways far more critical use cases to get right.

We think it's key to treat exception handling as something first-class in relationship to the rest of the framework.

TIBET source contains extensive debugging, logging, and assertion hooks to make it easier to craft enterprise-quality web applications.

Most JS has virtually zero error handling code.

We think a framework has to do a _lot_ better.

Exceptions are perhaps even more critical use cases to get right.

Error-handling code should be a first-class use case, particularly in an enterprise framework.

Humans Make Poor Compilers And Poor Interpreters

Hand-compiled code is part of the culture for JS.

For some reason the JavaScript community took a long time to build tools to minify scripts for transmission over the wire. In the interim a culture of hand-compiling JS code arose, ostensibly to keep bandwidth requirements small so applications would load faster.

Well, humans make poor compilers and poor interpreters.

With minifiers and tools that culture needs to change.

Now that there are plenty of minifiers and other optimization tools for JavaScript there's no rational reason to write source code that's obscure when there's a more readable and maintainable way to write it.

People are only impressed until the first bug.

People are only impressed with small and terse code until they have to decipher it under pressure to fix bugs. And there are always bugs.

Our focus is on writing code those who've never seen TIBET can use and maintain without us.

We write in a more maintainable style, "redundant" comments and all, with a focus on writing code that could be maintained by a new JS developer with no experience with TIBET and no access to its authors.

Your Coding Standard Is The One Your Tools Enforce

No coding standard survives its first encounter with a team -- unless it's enforced by an automated tool ;)

With all the arguments for and against different coding standards the one thing we've learned over time is that no coding standard survives real-world development -- unless it's enforced by an automated tool.

Tools like ESLint are excellent starting points for ensuring that whatever coding standard you choose for your applications you'll get a consistent codebase independent of how many developers are involved.

Automated checks let developers focus on the real issues in code review.

The nice thing about enforcing the lower-level elements of your coding standard via tools is it lets developers focus on higher-level concerns like API consistency, naming conventions and, dare we say, comments instead of fighting over where the whitespace and braces belong.

We want those checks to happen incrementally.

With TIBET our intent is to provide tools which let your developers work interactively while taking advantage of incremental standards checking.

Our Priorities

Design is a discipline defined by making intelligent trade-offs as you try to solve for all your competing goals.

Design is about making consistent, priority-driven tradeoffs.

If you're going to make design trade-offs consistently enough that your resulting design is coherent you need a set of clearly defined priorities that determine which design wins when the inevitable conflicts between your design goals arise.

To make those trade-offs you need to know your priorities.

For some projects the key priority seems to be "Smaller is better."

For some projects it's "Startup speed trumps everything."

For some it's apparently "Give me functions or give me death."

TIBET is decidedly different.

TIBET's top priority, the filter that trumps all other considerations, is "Must run offline."

TIBET's priority is "Must run offline".

Must Run Offline

Running offline might seem like an obscure goal that applies to only a few applications, but saying "runs offline" is the most succinct way of saying "Runs without server or network", an important forcing function.

That might seem obscure but it's a succinct way of expressing a lot of goals.

Running entirely in the client forces us to stay true to a balanced client/server architecture, forces us to stay server-agnostic, and forces us to create a fully-functional framework, one that can support porting your app to the client without any hidden gaps in functionality.

Server-agnostic, fully-functional, Client/SOA.

Must Use Native Web Technology

As earlier sections have pointed out, browsers understand HTML, XML, CSS, and JavaScript/JSON. That's about it. Any other technology would require us to rely on plugins, browser extensions, and other things which would compromise one of the key features of the web -- zero install. We're strict about keeping TIBET fully-functional without any add-ons.

No installs of any kind other than the browser itself. That's our goal.

Should Be Maintainable Year After Year

We target applications that deploy over 5-10 year lifespans. Those requirements are special.

Our target applications typically deploy and then are maintained for years, often 5-10 years. When an application is expected to live that long it's critical that the underlying framework and foundation be flexible enough, open enough, and extensible enough to survive over time.

Should Support Interactive/Immersive Development

Our vision is to run in the browser full time, with no reloads, in a fully-immersive and interactive fashion.

Our vision for TIBET has always been to recreate the fully-immersive development model of Smalltalk. As a result we see tools, particularly those that support interactive, immersive development, as key. Coupling that vision with our background building UI over CLI (both SQL and UNIX) means we've created both a command line and an IDE above it.

May Support Execution Within A Server-Side Container.

Oh, and running on the server side would be nice.

With the rise of Node.js and other server-side JavaScript containers we do believe there is a place for a server-side TIBET at some point. That said, it's not a high priority (hence May rather than Should or Must). Remaining server-agnostic will always be a higher priority than dependence on a TIBET-specific Server, at least for production use.

Our Stretch Goals (BHAGS)

Go BHAG or Go Home.

TIBET was designed with some Big Hairy Audacious Goals or BHAGs as Tom Peters would say.

Develop live. In Browser. Seamlessly offline/online. Zero-server. 5-10 years. With 80% markup.

In short, we think you should be able to develop live, in the browser, while your application runs; be able to support seamless online/offline operation; be able to build zero-server apps; have app life-cycles of five-plus years; and do it all using 80% markup and only 20% code.

Maybe we're a little crazy but setting the bar low doesn't inspire change.

Make those work and you have changed the rules.

In the end that's perhaps the key point. We believe change is needed; that we'll never build all the web apps needed without a shift in strategy.

Summary

So what's the essence, the Zen of TIBET?

It's using every ounce of compute power in the client to give the user the best possible experience, one that rarely waits on a server.

It's using JavaScript to support markup and CSS, as originally intended, in a largely invisible yet critical support role, not standing center stage.

It's using classes, metadata, and reflection to create tools that will help web development scale to support the millions of applications we need.

It's using what's been learned over almost two decades in the web while remembering that reinvention isn't innovation and very little is truly new.

And it's about making web development so fun we can't wait to do more.

The Zen of TIBET?

Use everything the client device can provide...

Put markup/CSS first; use JS in a support role...

Use OO, metadata, and reflection extensively...

Stand on the shoulders of giants...

And keep it fun!!

Contact us for more information, or to discuss how we can assist you in making your web applications more efficient and productive.

Contact us for more at:

info@technicalpursuit.com