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

A P P L I E D S O F T W A R E A R C H I T E C T U R E

The Origins Of TIBET™

IN PURSUIT OF MAXIMIZING WEB USER AND DEVELOPER EFFICIENCY.

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.

The extreme constraints of that early project forced us to reevaluate all we thought we knew about what a browser could and couldn't do. In the end we suspended our disbelief and fully embraced the client-side.

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.

Over-reliance on server-side processing fosters huge inefficiencies.

A natural side-effect of what we learned about client-side capabilities back in 1997 is that we became painfully aware of how perceived limits and over-reliance on server-side processing fosters huge inefficiencies.

Over the intervening years we've worked on a number of similarly-scaled projects and each has reinforced those early lessons.

We set out to address these issues with the TIBET Platform.

The TIBET Bookstore

To quantify just how inefficient things really were we did a simple experiment.

To help us understand just how much TIBET and Client/SOA might be able to improve efficiency we started with a simple experiment.

We decided to implement our own version of a bookstore application, a common demo application type at the time [1].

[1] "Duke's bookstore" was Java's demo; Microsoft countered with a .NET demo.

We went online and ran a query for books with JavaScript in the title. Each page we got back contained 25 book entries formatted in HTML. The book data was ~5k per page; the rest of the page text was ~85k.

Given our result set data we wondered if we could load 15k of book data (3 pages worth) and some client-side merge logic without changing the size of the initial page load. If so we'd be able to format the first 3 pages in the client without users noticing the difference in terms of load time.

We built a bookstore which moved data and HTML merge logic from server to client...

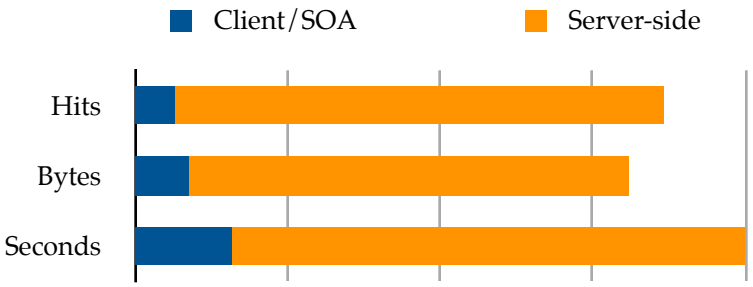
With experimentation we created a TIBET bookstore that did just that — reducing server hit rates and bandwidth by 66% over 3 pages by migrating the logic which merged book data with HTML to the client.

Then we took it further, responding to the initial query by sending the entire 60k of raw book data (12 pages worth) in the first result set.

Users who wanted to page back and forth, run drill-down queries, sort, filter, or perform other activities saw a larger initial delay...but that pause was offset by near-instantaneous response times for follow-on activities.

In the end, users who spent time searching, sorting, filtering, and reviewing the data over the equivalent of 12 pages reduced server hits by 91.6%, bandwidth by 86.1%, and wait time by almost 4 minutes (based on remote network speeds at the time).

... reducing server hits by 91.6%, bandwidth by 86.1%, and wait time by almost 4 minutes.



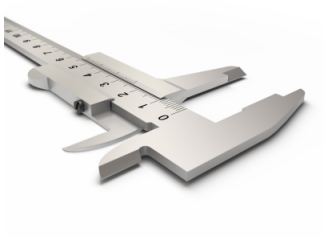
So what did our test, affectionately nicknamed Amazoom, teach us?

- There's a lot of room for improvement;
- focusing too heavily on initial load times is dangerous; and
- smaller isn't always better.

Measurement 101: Make it Better, not Smaller

"Load the first page" is not a use case.

Optimizing load time is a poor substitute for optimizing your users' key use cases end-to-end.



User's aren't done with your site when the first page loads, they're just getting started.

Our first lesson in measurement was that focusing on optimizing initial load time is a poor substitute for optimizing end-to-end.

By looking at a user's typical workflow we were able to reduce server hits and bandwidth by 66% — without altering user-perceived performance. That's an important lesson for any site, but it's a particularly important one for Enterprise sites which are more application than publication.

Conventional wisdom says smaller is better, but…

...it wasn't relevant whether the JavaScript we loaded was as small as possible, only that it was smaller than the content it replaced.

Our second lesson, a more subtle and profound one, was that it wasn't relevant whether the JavaScript we loaded was as small as possible, only that it was smaller than what it replaced.

In our experiment anything smaller than 75k would have met the goal of not altering initial load time. Because we focused on workflow efficiency rather than load time we felt free to use all 75K if we were adding functionality that would improve the user's efficiency and/or experience.

Smaller is, well, smaller. Whether smaller is better or worse depends on what's left out and what it could have done in support of the end user.

We took these lessons to heart when designing TIBET, focusing on better, not smaller, and measuring across the full end-user experience.

Measurement 201: Load Systems, not Silos

Conventional wisdom also fails to account for the impact of running multiple applications.

When users will be running multiple applications the mathematics of efficiency change again. And again there are a number of untapped areas for dramatically improving efficiency and performance.

The 1997 project we mentioned was one of roughly eighty (80) internal applications the customer wanted to port to the Web. Eighty.

Conventional wisdom, driven by the "minimize initial load time" mantra, recommends packaging each application individually. Unfortunately, optimizing subsystems often leads to suboptimal systems. When users run more than one application, each one is potentially a subsystem.

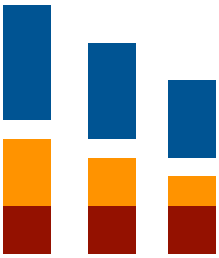
Optimizing subsystems often leads to suboptimal systems — and multiple applications in a suite are effectively subsystems.

Imagine an enterprise which has web-enabled its internal recruiting, employee evaluation, and vacation/holiday tracking systems. Let's further imagine our three applications rely on a common JavaScript framework and are 120k, 100k, and 80k in minified form — 300k total.

If each application is composed of 50% framework code then they have 60k, 50k, and 40k of framework code respectively. Clearly, some of that framework code is duplicated among those applications — perhaps 30k.

Each application will have different amounts of duplicated framework code, application-specific framework code, and custom code.

The stacked blocks below help illustrate our overall configuration. For each of three applications we have red (duplicate code) and orange (sharable code) blocks which represent unnecessary overhead:



Application-specific custom code.

Sharable application framework code.

Duplicated core framework code.

When each application is individually minified users load the duplicate chunks of framework code repeatedly as they switch applications (and parse it repeatedly with every distinct page load). Even if users load each application in a separate tab they'll load 60k they didn't need to — and using separate tabs eliminates the ability to share common data.

Loading and parsing duplicate code for each application and page degrades efficiency.

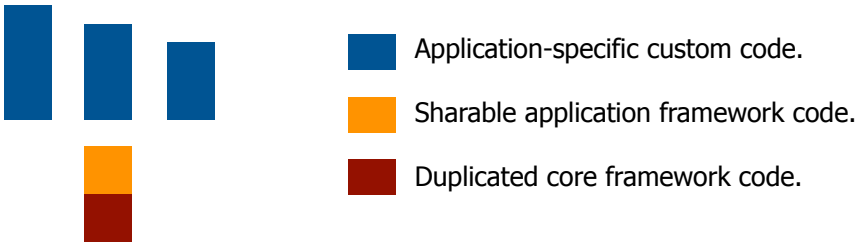
If we could load the duplicate/sharable framework code once and reuse it we'd get a minimum 20% improvement in load/bandwidth efficiency as well as eliminating redundant parse time with each unique page load.

TIBET shares code and data across applications, eliminating redundant load and parse times.

Avoiding HTTP overhead is the single most-effective thing you can do to optimize your web performance and the TIBET platform's architecture is designed to support the most parsimonious use of HTTP possible.

A single load and parse boot sequence caches the TIBET platform.

The TIBET platform "loads and stays resident" using a shared library architecture. In this approach a single copy of the platform is loaded and leveraged by all common application subsystems and modules in a suite:



Each application caches its code as well, reducing context-switch overhead.

As each application launches it verifies the library has booted, then loads any application-specific code. All loaded code is cached so later uses of that module happen with little or no HTTP or parsing overhead.

Configuration of TIBET packages is done using a custom manifest file approach which removes unnecessary dependencies from the source code. This allows reconfiguration and optimization of load packages to be done without having to edit any of the individual source code files.

The benefits of TIBET's unique shared kernel model increases with more applications.

When you're dealing with multiple applications in a larger enterprise suite the math ironically shows bigger kernels can be more efficient.

When done correctly the first module can efficiently load a kernel which represents the greatest common denominator of shared code rather than the least common denominator, or worse, only the silo'd code it needs. Remaining modules leverage that kernel to load significantly faster.

Measurement 301: Offline Maximizes Caching

No plugins or other extensions are required.

TIBET uses client-side logic to load and share code and data across application modules without plugins, browser extensions, or a server.

TIBET was built offline to remain server-agnostic and offline-ready.

To our knowledge TIBET is the only web framework built offline. TIBET's development team typically loads the framework from the file system, not a web server, ensuring the framework remains entirely server-agnostic and offline, dare-we-say "zero server", capable.

While running offline isn't always necessary it is often desirable. Workers who travel can attest to the fact that just because you have connectivity doesn't mean you want to use it. Connections can be intermittent, slow, costly, or a random combination of all these factors. Offline matters.

Just because you have a network connection doesn't mean you want to use it.

Where data is used across pages client-side caches can be leveraged to store changes to that data until it is ready to push to the server.

Perhaps the most compelling feature of persistent client-side caches is they can protect user data in the event of a network outage.

TIBET's caches also support fault-tolerant storage in event of a network outage.

What makes all this possible? Client/SOA.

Client/SOA: The Architecture Of Efficiency

Client/SOA is an architecture defined by moving four layers of functionality found in traditional web applications to the client:

- Interaction: handles direct interactions with the user
- Presentation: generates the interaction layer elements
- Application: processes page flow and validation logic
- Session: manages shared data between "commits"

Client/SOA moves the Interaction, Presentation, Application, and Session components to the client.

Three layers remain on the server:

- Authentication: certifies user credentials throughout a session
- Service: supports client compute and storage requests
- Storage: manages permanent storage/data access

Most Web 2.0 approaches (RIA/AJAX), push portions of the Presentation, Application, and Session layers to the client. Unfortunately, since portions of these layers remain on the server, scalability and fault-tolerance suffer.

Web 2.0 architectures offer a partial solution. Client/SOA takes it the rest of the way, maximizing efficiency.

There's no magic in why users are more efficient with Client/SOA. By moving logic as close to the user as possible there is simply less communication overhead. Users are more efficient as a result.

Local Throughput

Transactional workers rely heavily on keyboard support like shortcuts.

Mission-critical web applications often target transactional workers whose work involves high-throughput data entry. In these situations keyboard shortcuts are perhaps the most important contributor to user input efficiency.

TIBET generalizes these shortcuts into gestures which map to operations for easier development.

TIBET generalizes the concept of keyboard shortcuts into gestures, combinations of one or more events which can be mapped to meaningful operations. The simplest gesture might be a shortcut key like Ctrl-V mapping to Paste, but more complex mappings are possible.

Keyboard localization is supported for non-US keyboard layouts.

Another overlooked keyboard requirement is localization. A user can hardly be productive if his or her keyboard doesn't match the locale; TIBET supports fully-localizable mappings for non-US keyboards.

Mouse gestures build on native mouse events and offer extended control over the user interface.

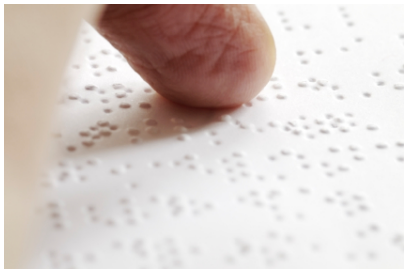
Mouse gestures, unique combinations of button and movement operations, are also supported in TIBET.

Gestures can also combine mouse and keyboard events if necessary.

Providing fine-grained device control is essential to interface efficiency.

Using gestures allows TIBET interfaces to be mapped for accessibility.

One of the most important aspects of having a generalized gesturing model is that operations initiated by devices are converted to events. As a result TIBET interfaces are easily mapped for accessibility.



UI widgets are also built for accessibility.

User interface widgets in TIBET are also designed with accessibility in mind, taking their cue from the W3C ARIA web standard.

Developer Efficiency

Applications that take too long to develop are applications at risk. Savings in infrastructure costs and increases in user throughput that could improve your bottom line aren't relevant until the application ships.

Applications that take too long to develop are applications at risk.

A lot of things can affect development velocity, but some key ones are the availability of qualified personnel, the development process they have to use, and the tools and documentation available to assist them.

Staffing, process, and tools all affect velocity.

Sadly, the dirty little secret of AJAX is that hiring is a problem. In his May 2006 AJAX Experience keynote, Scott Dietzen, former CTO of BEA and then CTO of Zimbra stated the hardest AJAX problem was recruiting. In all, Zimbra performed 400 interviews to find 10 AJAX developers. In the 7 years that have passed, hiring is still the hardest AJAX problem.

The dirty little secret of AJAX is that hiring is a serious problem, with rejection rates of 40-to-1 or more.

If finding talent weren't challenging enough, the development process for web applications is far from efficient.

One hidden drain on developer productivity is the need to click Reload. During active development a developer may click Reload hundreds of times, with each reload cycle potentially requiring that the developer re-authenticate, re-navigate, and re-enter data. It has a devastating effect on developer efficiency.

Even with qualified developers, the need to click Reload dramatically lowers productivity .

Browser inconsistencies — frustrating differences between browser vendors and/or browser versions with respect to HTML, CSS, or JavaScript — slow development even further.

Browser inconsistencies are another productivity time sink.

We took all of these issues into account, taking our cues from Smalltalk, Apple's Interface Builder, the UNIX shell, and other expressive, powerful, and productive environments we enjoyed before moving to the Web.

We designed TIBET to address these issues, from hiring qualified developers to making them more efficient.

We designed TIBET to be something you not only want to use, we designed it to be something you could consistently hire developers for.

The Authoring Pyramid

There's no shortage of solid HTML and CSS developers, but there are few JavaScript Gurus.

There doesn't seem to be a shortage of developers with HTML and CSS skills — unfortunately, the number of JavaScript developers ready to tackle mission-critical applications is far below the number needed if we're going to port our critical business applications to the web.

Experience shows this developer ratio can be 40-to-1 or higher, so it's key that TIBET not require skills that aren't in scalable supply.

Based on Zimbra's experience, which mirrors our observations, for every 100 HTML/CSS developers there are only 1 or 2 true JavaScript gurus.

TIBET reflects the proportions of this authoring pyramid in its design.

TIBET's authoring model is tag-based to take advantage of the large talent pool with markup and CSS skills.

For maximum scalability in terms of developer skills we made tags the primary interface. By choosing tag-based authoring we ensure content conforms to certain rules, much like with a compiled language.

Behind the markup interface of TIBET lies a layer of JavaScript that every developer will be comfortable with should they encounter it during debugging or other development tasks. This layer lets developers focus on URIs, Nodes, and other high-level objects common in the Web.

When complex work is required there are hundreds of Types and thousands of methods.

If it becomes necessary to create a custom widget or dive into more serious JavaScript work, TIBET provides hundreds of pre-built Types and thousands of methods to leverage, all organized around a set of strict naming conventions to assist with learning.

Custom widgets and other JavaScript code can be written by a very small group and published as custom tags for reuse.

Armed with TIBET, one JavaScript guru creating widgets, actions, and visual styles can support a much larger team of UI developers by publishing custom tags — tags which can be reused enterprise-wide.

Of course, regardless of how well we've maximized leverage for key developers, we still don't want them wasting their time clicking Reload.

Escaping Reload Hell

Our earliest frustration back in 1997 was the lack of tooling to allow us to interact with the application's HTML, CSS, and JavaScript directly.

Editing a file on the server, re-deploying it to the application server, reloading the page, and re-entering data to see if we'd fixed something seemed like programming using a remote control...with weak batteries.

Most web development feels like programming via remote control….a remote control with weak batteries.

Prior to our work in the Web we'd been developing in Smalltalk, a language where you manipulate the application while you run it. In that environment you could edit the user interface, or change how the application responded to a mouse click without ever reloading.

We remembered our time with Smalltalk as being much more productive.

We wanted that kind of fluid development environment for TIBET.

Our first step was to build an interactive command line tool, much like the UNIX or DOS command line. This allowed us to enter JavaScript for execution against our running application. Already we could see a dramatic shift. Rather than experimenting by editing a file and reloading we could now experiment endlessly with few, if any, reload cycles.

Our first step was to create a command line much like UNIX/DOS which dramatically reduced reload counts.

To access the more visual parts of the application we added a special context (or right-click) menu we called the "halo". Right-clicking on an element gave us control over it. Each of our tools are finding their way into a unified IDE which maximizes client-side development workflow.

Each of our tools is written in JavaScript; each is capable of being run client-side with no server to preserve our ability to work offline and to remain server-agnostic.

More tools are on the way as we move more and more into developing entirely in the browser.

Having addressed the majority of JavaScript-specific issues related to reload and efficient client-side development we next had to address the other browser technologies that are integral parts of web development.

The Three-Legged Stool

There are 3 technologies a browser understands, but understanding differs from browser to browser.

There are three technologies browsers natively understand: CSS, markup (HTML/XHTML/XML), and JavaScript. The problem is different browsers understand them a little differently, creating serious inefficiencies.

While virtually all AJAX libraries address inconsistencies in JavaScript, virtually none address problems with CSS or markup processing. We find that interesting, since the biggest complaints about browsers are rarely due to their JavaScript engines being buggy, their due to buggy layout.

With TIBET we wanted to control all three so we could fully support web developer's needs.

With TIBET we took control of all three native technologies, creating a three-legged stool which supports developers by resolving inconsistencies and offering forward-looking standards support.

The three legs of our stool are the TIBET library and our markup and CSS pre-processors.

The TIBET library adds OO features, logging, job control, and other modern language elements to Javascript.

The TIBET library includes a distinctly object-oriented core which makes it extremely easy to extend and maintain. Woven into this core are extensions for exception handling, logging, job control, HTTP(S), WebDAV, file access, XSLT processing, HTML templates, and more.

TIBET's tag processor treats tags as macros, and allows you to create your own custom tags.

The Tag Processor is TIBET's markup engine. Whenever tags are sent to the user interface TIBET checks each one to see if it maps to a macro. In that sense, the tag processor is a macro processor, expanding each tag and performing actions that tag's corresponding class demands.

TIBET's CSS processor helps ensure consistent UI rendering and other CSS tooling.

The CSS Processor is TIBET's CSS engine. All CSS used in TIBET passes through the CSS processor, which expands it (if you were using LESS etc) and makes whatever adjustments are necessary to help ensure consistent UI layout and forward-looking standards support.

TIBET is unique in encapsulating not only JavaScript, but XML and CSS. Some of this is done simply to handle browser differences. Some is done to support in-browser authoring more efficiently via metadata.

Organizational Efficiency

Organizational efficiency is the final frontier.

There's one last area of efficiency to discuss — organizational efficiency.

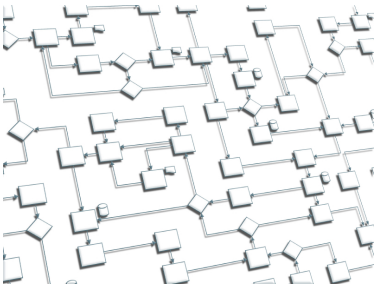
There are two areas of organizational inefficiency we wanted to address with TIBET. First, we wanted to decouple the client and server teams so they could work in parallel. Second, we wanted to improve the communication and collaboration options in web-based applications.

Parallel development is easily supported in TIBET, unblocking client-side developers.

Decoupling the client and server teams is a key benefit of Client/SOA. In a Client/SOA architecture all client and server communication occurs via web service calls. As a result, once the two teams agree on common APIs and data formats they can develop modules independently using mock data, the same mock data used for unit testing the two sides.

XMPP support is built in and integrates seamlessly into the UI.

To help improve communication and collaboration in web applications, we added XMPP[2] support in 2002.



Our goal with XMPP wasn't to create a web-based chat client — it was to workflow-enable web applications.

Presence and message elements streamline business communication.

As an example, imagine a purchasing application where the agent's view includes presence icons for the requester and manager with purchasing authority. Clicking a presence icon could open inline chat to that user, streamlining communication and increasing Enterprise-wide throughput.

Pub/sub events provide the foundation for true workflow — all the way out to the browser.

Alternatively, imagine a document-sharing application where users need to be aware of changes made by other team members. We created a straw-man document-sharing application using Amazon's S3 service to store the documents, Amazon's EC2 service to host an out-of-the-box XMPP server, and one page of TIBET markup in less than a day using no server-side code whatsoever. To us that's the ultimate validation of Client/SOA and TIBET as well as the true power of modern browsers.

---

[2] XMPP is an IETF standard XML Messaging and Presence Protocol.

Summary

Efficiency is job #1.

Our goal is to increase user, developer, and organizational efficiency with respect to web application design, development, and deployment.

Focus on architecture and authoring model.

Compelling changes in efficiency come from two main areas: architecture and authoring model. Additional improvements in efficiency can also be made by managing user input devices to maximize local throughput.

Measure end-to-end, and consider the impact of application suites.

Conventional wisdom around web development does not reflect the importance of end-to-end performance measurement or the impact that running multiple applications should have on web architecture.

Think client/server, or at least Client/SOA.

Changing the application architecture to a Client/SOA model directly affects the efficiency of the user by moving logic as close to the user as possible. This is simply client/server principles fully applied to the Web.

Target a scalable pool of developer talent and feed them from one or two Gurus as needed.

Focusing on tag-based authoring to leverage a more scalable talent pool. Providing those developers with tools to avoid reload overhead and encapsulating all three browser technologies rather than just one dramatically increases the efficiency of the development process/team.

Connect everyone, then help them collaborate.

Integrated XMPP support provides the infrastructure needed to presence-enable your enterprise applications and make them workflow-aware. The resulting potential impact on your organization may well outweigh all the other improvements combined.

Contact us for more at:

info@technicalpursuit.com

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