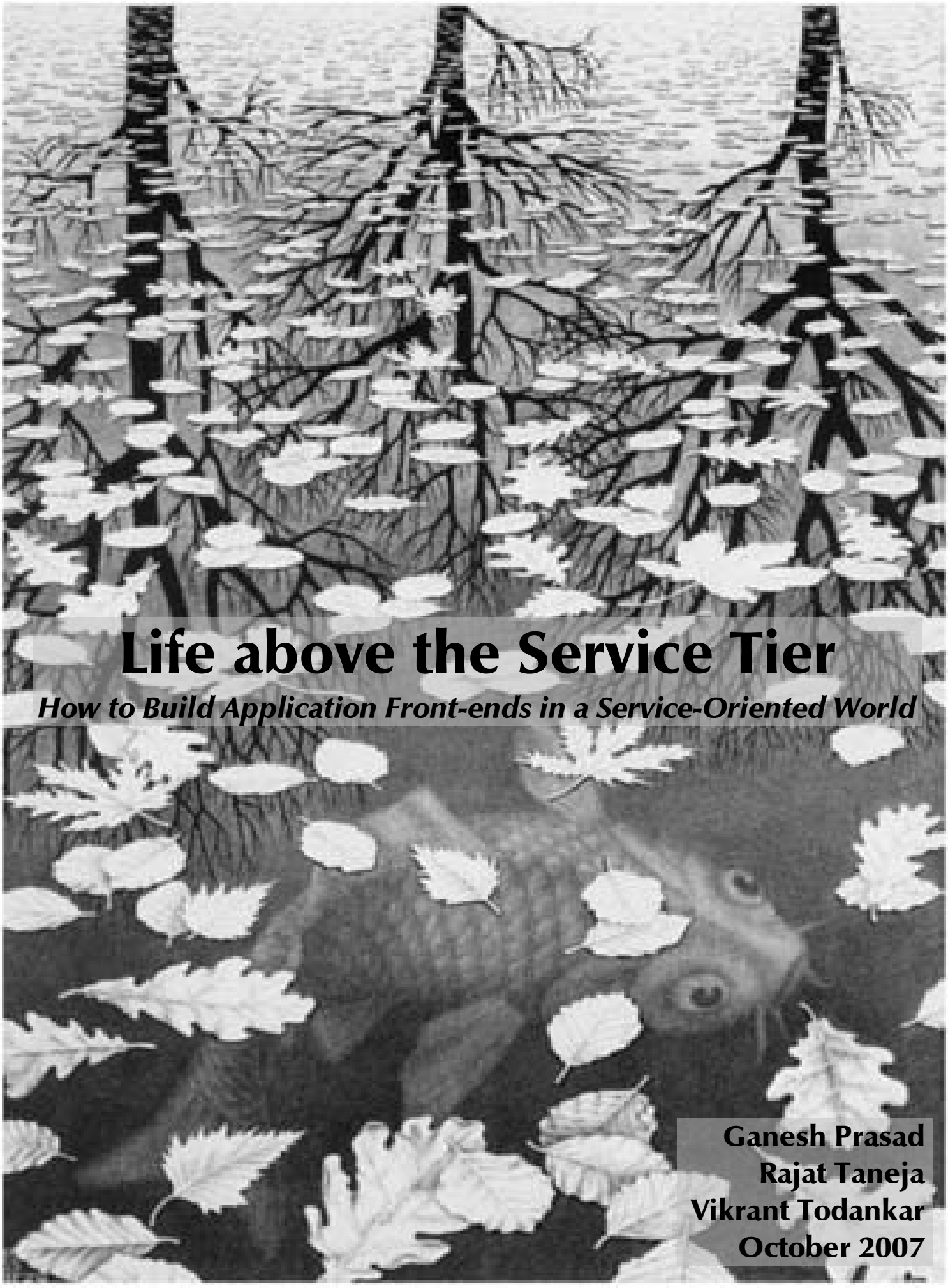


Life above the Service Tier

How to Build Application Front-ends in a Service-Oriented World

Ganesh Prasad
Rajat Taneja
Vikrant Todankar
October 2007

"Life above the Service Tier"

Dedication

To innovators like Tim Berners-Lee and the Oak (Green project) team at Sun Microsystems who created the core foundations upon which so much of our digital civilisation depends (The Web and Java);

to visionaries like Trygve Reenskaug, Roy Fielding and Jesse James Garrett who identified fundamental patterns and named them for the first time (Model-View-Controller, REST and AJAX);

to practitioners like N. Alex Rupp and Michael Jouravlev for easing our way with insights, tips and techniques (The WARS architectural style and the POST-Redirect-GET pattern);

to Firefox and Apache for keeping the Web open – at both ends;

to Microsoft, for a Snape-like good deed – XmlHttpRequest;

in short, to all those giants on whose shoulders we stand today.

Sydney, October 2007

Cover illustration: "Three Worlds", M.C.Escher, 1955 Lithograph

Synopsis

How do we design and build the Presentation Tier of an application in an increasingly service-oriented world? We believe there is a definite answer, although it is not a particular technology but rather an *Architectural Style*. We call this style **SOFEA**, for Service-Oriented Front-End Architecture.

The principles of SOFEA are:

0. Decouple the three orthogonal Presentation Tier processes of **Application Download**, **Presentation Flow** and **Data Interchange**. This is the foundational principle of SOFEA. (Interestingly, the most common Presentation Tier technology, i.e., traditional web technology or "Web 1.0", fails this principle.)
1. Explore various **Application Download** options to exploit usefully contrary trade-offs around client footprint, startup time, offline capability and a number of security-related parameters. (The key differences between "thin" and "rich" clients lie in these trade-offs, and therefore SOFEA is a metamodel for both types of applications.)
2. **Presentation Flow** must be driven by a client-side component and *never* by a server-side component. Client state must be managed within the client. (We show that the Front Controller "pattern" represented by all server-side web frameworks is in fact an anti-pattern, which is why there are so many variants of it and why none of them satisfies.)
3. **Data Interchange** between the Presentation Tier and the Service Tier must not become the weakest link in the end-to-end application chain of data integrity. The Presentation Tier must support equally rich data structures, data types and data constraints. (In this regard, the inherent weakness of "Web 1.0" makes it hard to integrate with the Service Tier. We recommend the use of XML as the common data denominator for the two tiers). Ideally also, the Data Interchange pattern between the two tiers should follow the peer-to-peer model rather than the client/server model to enable more natural event notification.
4. Model-View-Controller (MVC) is a good pattern to use to build the Presentation Tier. (This is not to be confused with Front Controller, which is an anti-pattern.) The MVC Controller is the key front-end component which manages client state and drives both Presentation Flow and Data Interchange processes.

There are many Presentation Tier technologies available to developers today, including some that were very recently announced. While all of them inherently allow wide latitude in the way they are used, we believe that adherence to SOFEA principles will ease their integration into an increasingly SOA-oriented enterprise infrastructure.

Table of Contents

Part I – The Way Things Are

Introduction

Before we settled on the subtitle that you see on the title page of this paper, we debated many other candidates:

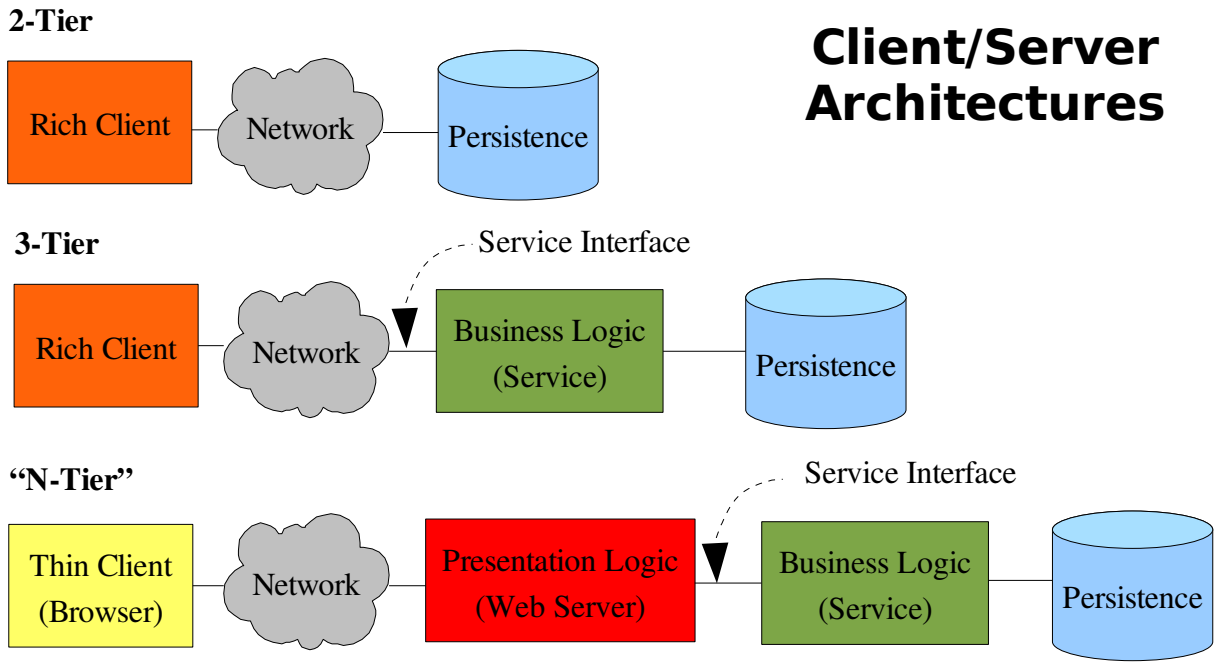
- An Architectural Model for AJAX
- Why are There So Many *$#@&(%^! Web Frameworks?
- Unifying the Thin and Rich Client Models
- Extending SOA to the Presentation Tier
  etc.

This paper is about all of these topics, but it is ultimately about providing architectural guidance around building *application front-ends* (i.e., the Presentation Tier), because that is the one area that the SOA (Service-Oriented Architecture) revolution seems to have left behind.

Let's start from some basic principles.

Application Architectures

The application architectures we use today are all generally some variant of Client/Server, as shown below.



(We have omitted the Domain Model component between Business Logic and Persistence because it is not important to the topic we want to discuss here.)

Note the Service Interface referred to in this diagram. We believe that is the crucial interface for applications. There is a lot of intellectual ferment driving innovation below this interface (exemplified by the term SOA), and we think that story is coming together rather well. There is also a lot of innovation happening *above* this interface, but those efforts seem to us to be a bit haphazard with no overarching vision or direction. We don't see these two worlds meshing together gracefully, which is why we wrote this paper. We believe we have an end-to-end architectural solution for application builders, and it deals with fixing what's above the service interface.

But first, what is service orientation and what does it mean for application design and development?

Service-Oriented Architecture (SOA)

SOA is a buzzword with a lot of associated hype, but it shouldn't be discounted for that reason. It is our firm belief that SOA is of significant importance to organisations. The ones who do it well will reduce their costs and improve their flexibility. To those readers who don't understand the term, here is our explanation in a nutshell:

SOA is about loose coupling between systems. Loose coupling means eliminating implicit dependencies between systems and stating all legitimate dependencies in the form of explicit contracts between them. (The crucial terms have been underlined.)

It sounds simple, but the devil is in the details. In the non-technology world, we can readily see the benefits of the SOA way of doing things. Companies that partner together should sign clear legal contracts rather than (say) relying on the fact that their CEOs are first cousins (What happens to the hundreds of interactions between the companies if one of the CEOs gets hit by a bus, ten years after their private arrangement, and a less accommodating replacement takes over?).

We're generally capable of recognising and avoiding implicit dependencies when we see them in the non-technology world, but we commit such blunders all the time in the technology world. (E.g., "You're using Technology X and we're using Technology X too, so it should be easy for us to integrate!") That's an *implicit dependency*, and therefore, even if Technology X is an integration suite like Sonic's or TIBCO's, *that's still not SOA!* SOA is about *explicit* contracts, and probably should have been called COA (Contract-Oriented Architecture). It would have made it easier to understand.

Some of the aspects of a contract deal with "plumbing" (data formats, encoding, wire protocols, etc.), but one of the many SOA pitfalls is in thinking the plumbing is all there is to defining contracts and implementing SOA. A large part of a contract (probably even a larger part in terms of effort) lies in the business space. Vocabulary is a prime example. What are the "nouns" and "verbs" of the business? If one division calls customers "customers" and another division calls them "clients", clearly more is required to integrate the operations of these divisions than some technology plumbing.

Data is important, as we will emphasise again and again. Data types, data structures, data value constraints, *taxonomies* - don't embark on SOA without them. Technology will solve a part of the problem. It will standardise wire protocols (e.g., HTTP). It will standardise data encoding (e.g., Unicode) and data formats (e.g., XML and XML-derived languages). It will even standardise Message Exchange Patterns (synchronous request/response, asynchronous request/response, one-way messaging, publish/subscribe) and Qualities of Service (security, reliability, transactionality). But technology cannot standardise business vocabulary or rationalise business processes.

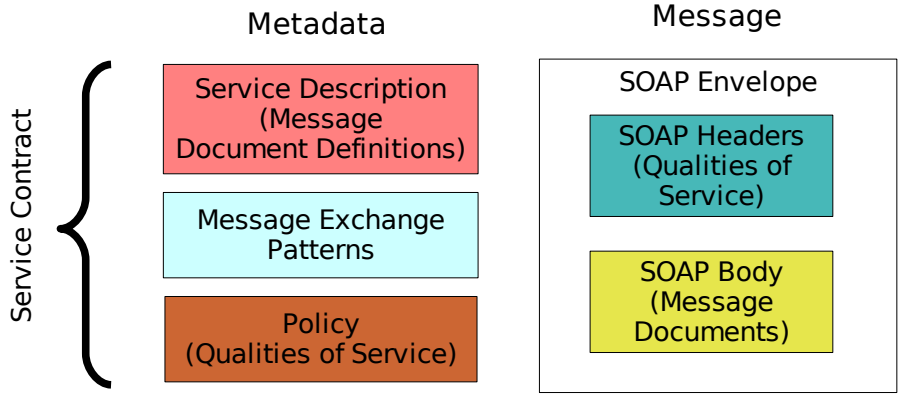
While the world is still at a relative state of immaturity with regard to SOA, all indications are that it is headed the right way. We don't believe the SOA direction needs correcting, but we recognise that it will take a few years for best practice to evolve and its benefits to be realised by the mainstream.

There are two more-or-less competing approaches to SOA, and they are “SOAP-based Web Services” and “REST”.

SOAP-based Web Services

Many people still associate SOAP with XML-based Remote Procedure Calls, but SOAP-based Web Services technology has outgrown its RPC roots to emerge into a flexible and powerful messaging model (the “document/literal” style). This approach to SOA assumes that all systems are independent “peers”. No system has any knowledge of the internals of another or any control over the functioning of another. The most such systems can do is send messages to one another and hope they will be acted upon. Systems publish contracts that they undertake to honour, and other systems rely upon these contracts to exchange messages with them.

Contracts between systems are collectively called metadata, and comprise service descriptions, the message exchange patterns supported and the policies governing qualities of service (a service may need to be encrypted, reliably delivered, etc.) A service description, in turn, is a detailed specification of the data (message documents) that will be sent and received by the system. The documents are described using an XML description language like XML Schema Definition. As long as all systems honour their published contracts, they can interoperate in a more or less guaranteed way, and changes to the internals of systems never affect any other. Every system is responsible for translating its own internal implementations to and from its contracts.



In short, the SOAP-based Web Services model views the world as an ecosystem of co-equal peers that cannot control each other, but have to work together by honouring published contracts. It's a valid model of the messy real world, and the metadata-based contracts form the SOAP Service Interface.

REST

REST stands for REpresentational State Transfer. Although REST's proponents claim that it is neither more nor less than an *architectural style*, it ends up being more concrete than that, because REST also has a *de facto* wire protocol for data interchange – HTTP.

Basically, the thesis of REST is that all distinct resources on the web that are part of the "universe of discourse" are (or should be) uniquely identifiable by a URL. All operations that can be performed on these resources can be described by a limited set of verbs (the "CRUD" verbs) which in turn map to HTTP verbs. The following table shows this mapping:

Generic verb (CRUD)	HTTP verb
Create	POST
Retrieve (also Search/Find)	GET
Update	PUT
Delete	DELETE

REST provides a complete and universal logical protocol for ready use by applications. The physical protocol is, of course, HTTP. While REST's proponents claim it to be much less "heavyweight" than its SOAP-based Web Services rival, this is not an easy shortcut for the lazy developer. REST emphasises proper taxonomy just as much, and supports XML-based documents to a similar degree. Even in REST, representations are decoupled from actual implementations. The combinations of supported HTTP verbs and resource URLs form public contracts that systems will honour, regardless of how they are implemented.

Thus, the REST approach accomplishes loose coupling between systems using the standard vocabulary of the web – HTTP verbs and URL nouns, a "contract" that hides actual implementations behind a common façade – the REST Service Interface.

We like both approaches as we think they're both well thought out. We don't believe that either of them will "win". Both seem set to gain in popularity and we believe both will continue to coexist. The implication for Presentation Tier technologies is that they must be able to interface with *both* styles, not just either one, because organisations are likely to have legacy services built using both.

That is the state of play below the Service Interface. What does the world look like above it?

The Components of a "Client" (User-Facing) Application

Whenever we use a client device to run an application (and this client device could be a workstation or any of the newer handheld devices we all own and love), three separate and distinct processes take place, whether we realise it or not:

1. Application Download (AD)
2. Presentation Flow (PF)
3. Data Interchange (DI)

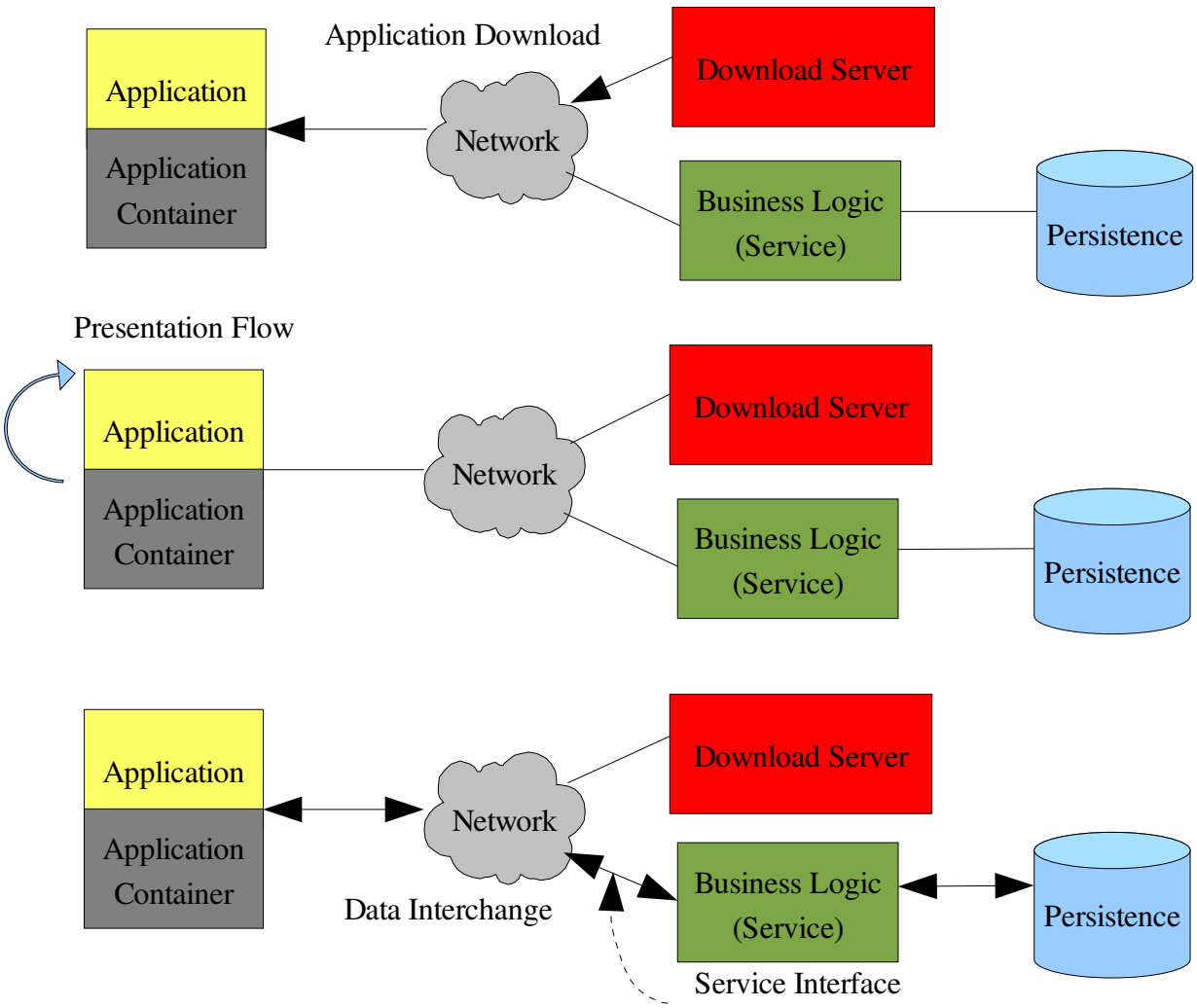
The application we run isn't part of the client device, even if it came bundled with it. Somebody put it there for us to run. We may have installed this application ourselves from a CD or downloaded it from a website, or we may have simply clicked on a hyperlink in a browser and arrived at its starting page. In any case, an "Application Download" (AD) has taken place. We cannot run an application on a client device without this step.

As we work our way through the application, the screen in front of us visibly changes, not just in terms of the data it displays, but in its very structure. When these changes are dramatic enough, it creates the illusion that the entire screen has been replaced by another one. Let's call this phenomenon "Presentation Flow" (PF). We won't call it "Screen Flow" because it is equally applicable to non-visual user interfaces like the ones used by phone- or voice-based applications. For the time being, let's not worry about how Presentation Flow is actually implemented, but be forewarned that this will be a major focus of our paper.

Be all that as it may, the ultimate purpose of our using the application is to manipulate data in some manner. Our client-based application is either creating, searching for, retrieving, updating or deleting data somewhere. An application that has nothing to do with data is inconceivable. In fact, we could argue that the ultimate purpose of any non-trivial application is "Data Interchange" (DI), and that the other two features (Application Download and Presentation Flow) are just means to this end. Of the three, Data Interchange is the only one that has anything to do with the Service Interface.

As a further observation, all client-side applications require a "container" of some description to run, whether they're described as "thin client" or "rich client". With "native" rich clients such as Visual Basic applications, that container is the operating system itself. Cross-platform clients such as Java applications need a container called the Java Virtual Machine. And as we said, even thin client applications need a container. The browser is their container, a combination of a rendering engine for HTML and a runtime engine for JavaScript. Applications built with technologies like Flash usually use a special browser-based plugin as their container, so that's a container within a container. The container may need to be deployed onto the client device before the application is downloaded.

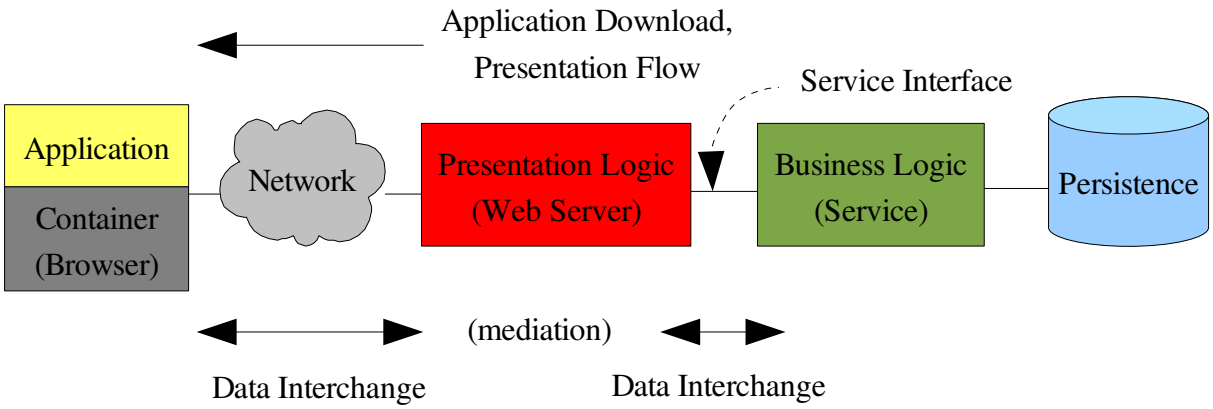
The following diagram illustrates the above points within a common context.



This is a logical model. With “thin client” or web applications, the component that we call the Download Server in the above diagram plays a role in Presentation Flow and Data Interchange as well. This is, of course, the web server.

Web servers usually drive Presentation Flow in thin client applications, and a number of web frameworks have sprung up to augment basic web servers with such capability. Web servers also act as *intermediaries* between the Client and Business Logic during Data Interchange operations, often caching session data and performing other optimisations.

The following diagram illustrates the role that web servers traditionally play in all three processes of an application front-end.



We will argue that this multiplicity of web server roles is a serious problem affecting end-to-end application architecture and impacting our ability to realise the full benefits of SOA. But let's first examine the differences between “thin” and “rich” to understand exactly how they're different, if at all.

The Thin and Rich Client Models

Thin and rich client applications have been with us for so long that we take their separateness as axiomatic. However, we will show that they are based on fundamentally the same model, and that what makes them different is a simple (albeit profound) architectural choice.

If we examine thin and rich clients according to the three application features we identified in the last section, we see some interesting differences:

Feature	Thin Client	Rich Client
Application Download (AD)	Application Download is not a separate step. It occurs "on demand" as part of Presentation Flow.	In the straightforward implementation of a Rich Client, Application Download is an explicit operation and results in the entire application being downloaded at once before execution can begin. Updates are fresh downloads. "Smart clients" are those rich clients that make these processes less painful, e.g., browser-launched automated installs, incremental updates, etc.
Presentation Flow (PF)	Presentation Flow may occur locally on the client (using JavaScript magic), but is more commonly driven from the server side. Many web frameworks exist to bring structure to this latter process, most using a design pattern called "Front Controller", which is a pseudo-MVC framework.	Presentation Flow is managed entirely locally by the downloaded application with no assistance from any server side resource. The design pattern normally used is MVC ("Model-View-Controller").

(Table continues)

Feature	Thin Client	Rich Client
Data Interchange (DI)	Data Interchange between Client and Business Logic is not direct but *mediated* by a server-side Presentation component (a web server). Between client and web server, Data Interchange traditionally uses the HTML/HTTP model, i.e., data is sent from client to web server as GET or POST parameters (name-value pairs); data is pre-formatted with HTML markup when sent from web server to client. In either direction, data is not treated as a first-class entity with support for types, structures and constraints. Data Interchange follows request/response semantics.	Data Interchange between Client and Business Logic is usually direct, not mediated by intermediate Presentation components. Any number of wire protocols and data formats are used, including object serialisation. Of late, XML is emerging as a common standard for data formatting. Data Interchange generally follows request/response semantics. It is equally possible for rich clients to support peer-to-peer interactions, but this may require changes to firewall rules within corporate networks.

Thin and rich clients also exhibit some common superficial differences:

1. Rich clients are “rich” compared to thin clients because they feature more visual “widgets” and because they are more “interactive”. With the advent of AJAX in the thin client universe, however, both these distinctions have melted away.
2. Thin clients are “thin” compared to rich clients because they have a smaller “footprint”. In other words, a thin client application requires minimal time to start up because there is no separate download step. This distinction endures, and it is because of the architectural choice made by thin clients implementing Application Download as part of Presentation Flow.
3. Rich clients are capable of offline or disconnected use, but thin clients require to be “online”, or connected to a Download Server (web server). This distinction endures as well, again because of the thin client architectural choice regarding Application Download and Presentation Flow.

The Real Differences between the Thin and Rich Client Models

It would appear from our analysis that the enduring differences between thin client and rich client applications stem from the architectural choice made by thin clients to avoid a separate Application Download operation by making it a part of Presentation Flow. This choice provides a usefully contrary trade-off. Thin clients have a small footprint and start up quickly, but they cannot be used offline.

By and large, this choice has proven to be a far greater advantage than a disadvantage for thin clients, as evidenced by the explosion in web applications over the past two decades. Indeed, Presentation Flow in the thin client model can be considered to be nothing more than "Lazy Application Download" or "Application Download on Demand", which carries the respectability of a performance optimisation. Anyone who has waited for a web-based Java applet or a Flash application to load can appreciate the wisdom of this choice.

In other words, the lack of offline-capability in the thin-client model is the result of a conscious choice and should not be considered a flaw. However, thin clients do suffer from three serious architectural flaws, the discussion of which forms one of the central themes of this paper.

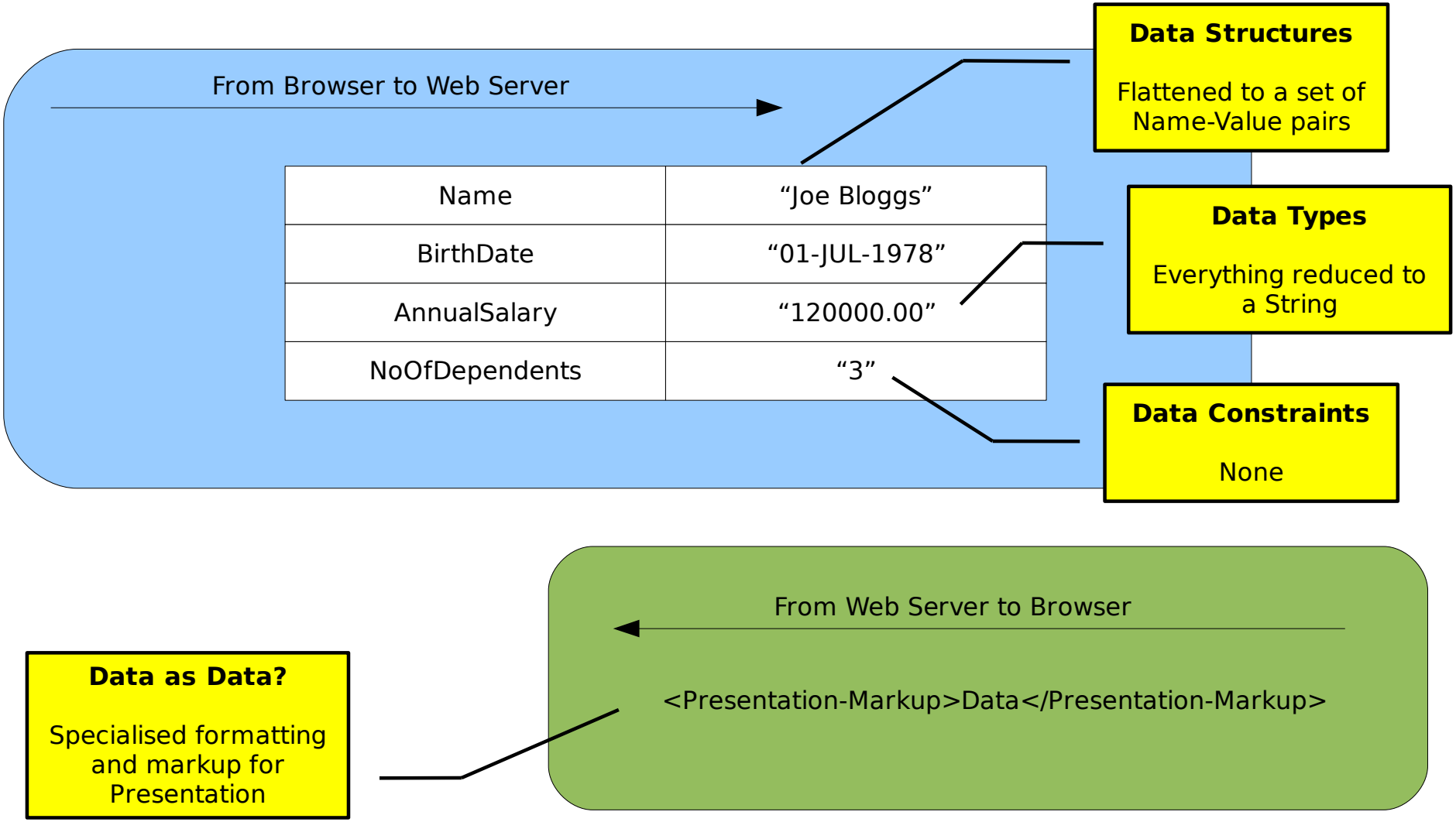
Architectural Flaws in the Thin Client Model

Flaw 1: No Mechanism to Ensure Data Integrity

There's a saying that if we want to judge someone's character, we need to observe how they treat those beneath them. Well, we find it useful to judge technologies based on how they treat data. From the way thin-client (web) technology treats data, it doesn't seem like a very nice technology to know.

The following diagram illustrates the web's problems with data.

The Web's Lack of Respect for Data



Consider the way data is sent from the browser to the web server.

GET parameters are simply tagged on to the end of a URL, following a question mark. Each parameter is a name-value pair. There are no data types other than "string". What about the relationship between different data items? There is no hierarchy. All data items are at the same level. What about constraints governing which values are valid and which are not? There are none. Any data can be transported in a field, regardless of how meaningless it is.

The situation is the same for POST parameters, except that they are mercifully not tagged onto the end of a URL in a visible way. Data is still stripped of its rich structures, types and constraints. There is consequently a lot of effort expended on the server side in putting Humpty Data together again, with needless validations and data conversions.

Consider next how data comes back from the web server to the browser. Data returned to the browser is "pre-cooked" for presentation, i.e., marked up with presentation-oriented tags.

This lack of respect for data *as data* is a fundamental characteristic of HTML-over-HTTP. It has been with us since the inception of the web, but no one seems to mind.

In a service-oriented world, this is a serious shortcoming because strong data definitions form a very big part of the process of service enablement. If the Presentation Tier does not adequately enforce data integrity, a lot of needless processing needs to occur near the Service Interface, where the two tiers meet. Wouldn't it be better if the Presentation Tier could adopt the data-related mechanisms, if not the actual data defnitions, that the Service Tier uses?

Flaw 2: Coupling of Presentation Flow and Data Interchange

The thin client approach of tying Application Download to Presentation Flow cannot be faulted. As we saw, it provides a usefully contrary trade-off compared to the rich client model – small footprint/quick startup versus offline capability.

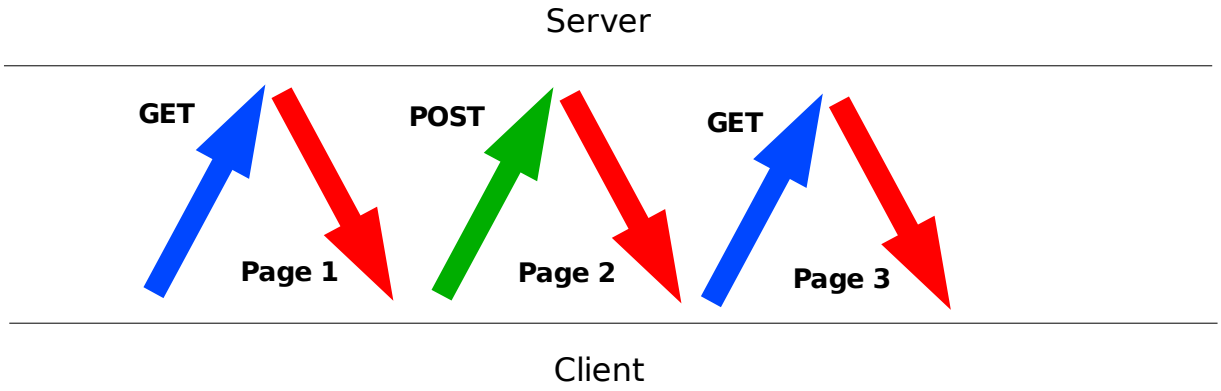
However, the other example of tight coupling (between Presentation Flow and Data Interchange) is the second major architectural flaw in the thin client model.

It is not possible to trigger a Presentation Flow in a web application without initiating a Data Interchange operation (e.g., a GET or a POST). More vexingly, every Data Interchange operation willy-nilly results in a Presentation Flow. It's a classic case of tight coupling between two orthogonal concerns. This tight coupling has not just been tolerated. Countless numbers of flawed applications have been built on this model, and they "work", albeit inelegantly. In hindsight, it might have been better if they had been completely broken, because then this aspect of the thin client architecture would have been fixed.

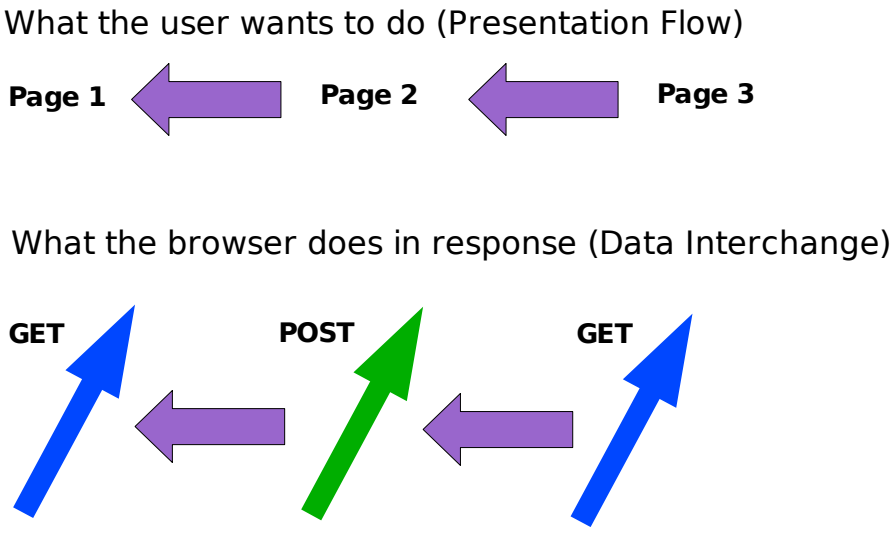
It's only in very recent times, and thanks to AJAX, that our collective eyes have been opened to the possibility that Presentation Flow and Data Interchange *can* be decoupled. AJAX seems like magic! Screens can display fresh data without blinking! We call this "greater interactivity" and praise it with high praise, whereas all that AJAX has really done is break the lockstep coupling between Presentation Flow and Data Interchange that should never have existed in the first place. (To be fair, a lot of the "rich interactivity" attributed to AJAX is thanks to plain DHTML. AJAX deals solely with Data Interchange. DHTML takes care of assorted visual magic, including Presentation Flow.)

No one seems to have pointed out this basic architectural problem or done anything about it for more than a decade, even though we have known about its major symptom for almost all of that time – the browser back-button problem.

Here's what a traditional (non-AJAX) web application looks like in terms of its Data Interchange and Presentation Flow steps:

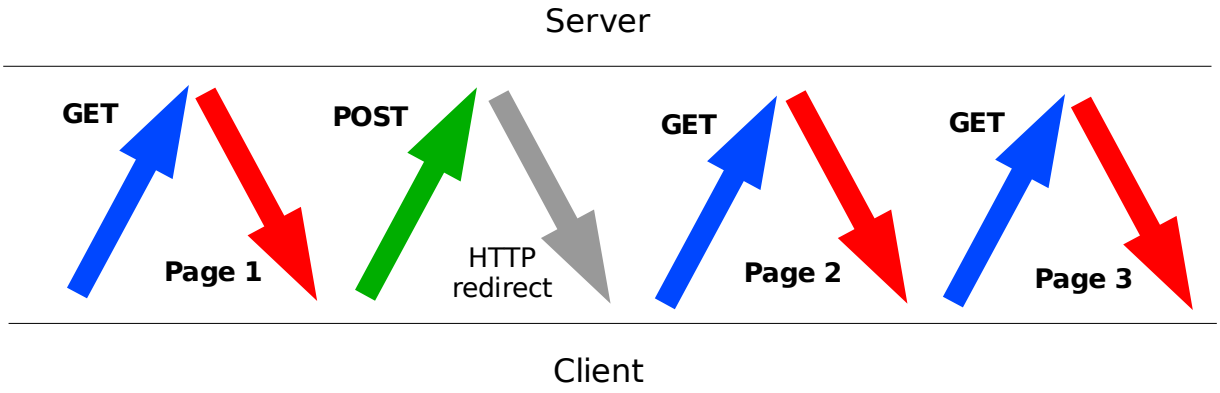


As can be seen, every page in the Presentation Flow (the downward facing red arrows) is in direct response to a Data Interchange request represented by the upward facing arrows (blue for GET, green for POST). If the user wants to go back to a previous page, the browser can only do so by re-issuing the Data Interchange request that resulted in that page, as shown below.

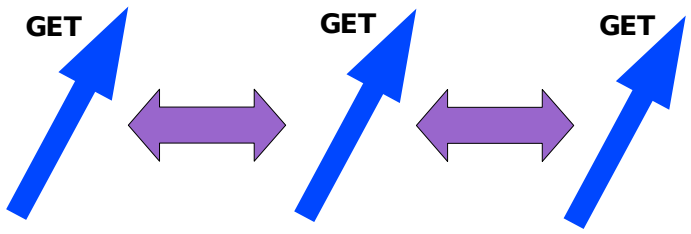


Therein lies the problem. POST requests, as the REST experts will tell us, is neither safe nor idempotent, so using the browser back button on a naively-designed web application will end up re-sending POST requests to the server and thereby cause considerable grief because of its side-effects on data. Modern browsers will warn the user whenever an attempt to go back a page will require a POST to be reissued, but that is not a solution. It's like a warning sign near an open manhole, but users may still ignore it and come to a sticky end.

The standard application design pattern used to avoid this problem is called POST-Redirect-GET, and it works like this:



Whenever the client issues a POST Data Interchange request, the web server does not respond with the next page in the Presentation Flow. It sends back an HTTP “Redirect” request to the browser, in effect asking it to issue a fresh GET request for that page. So all pages in the Presentation Flow are forced to be responses to GET operations, which are both safe and idempotent. A web application designed with the POST-Redirect-GET pattern is friendly to the browser navigation buttons. Moving backwards and forwards in the Presentation Flow only requires safe and idempotent GET requests to be sent to the server, as shown below.



Although ingenious, the POST-Redirect-GET pattern is mere band-aid over the fractured web architecture. The fundamental problem is the tight coupling between Presentation Flow and Data Interchange, and that's what needs to be fixed.

In a sense, the problem is unsolvable. The HTML-over-HTTP interaction model doesn't inherently work any other way. Web servers can only supply HTML pages (Presentation Flow steps) in response

to HTTP requests (Data Interchange operations). Besides, both Presentation Flow and Data Interchange logic are crucially influenced by a subtle aspect of the application – the "client state", which means that whichever component of the application holds the client state will act as a point of coupling for the two. And so these two factors lie at the root of the Presentation Flow/Data Interchange coupling problem.

Although the above flaw does not affect the interaction between the Presentation and Service Tiers, it is an internal problem of the Presentation Tier with its own history (see following box).

Incidentally, we believe we have the answer as to why there are so many server-side web frameworks to drive Presentation. Web frameworks are based on the Front Controller anti-pattern (which looks like a pattern only when compared with raw HTML-over-HTTP). Front Controller does not help in decoupling Presentation Flow from Data Interchange, and is therefore part of the problem. If we keep developing newer and "better" variants of an anti-pattern, is there any wonder none of them will satisfy? The correct approach is to repudiate this anti-pattern altogether. The historical reason for its invention no longer exists to justify its continued use.

History Detour

The history of the web application starts with Netscape. Netscape was perhaps the first company to realise that the browser itself was becoming a candidate for the role of application platform that the operating system hitherto played. Netscape talked about how applications would no longer care about which operating system they were running on, because all they needed was a browser.

Needless to say, such talk alarmed Microsoft a great deal, because the desktop operating system franchise was everything to them. What happened next was predictable. Microsoft released its own browser, Internet Explorer, bundling it free with every copy of Windows. Crucially, Internet Explorer was not completely compatible with Netscape's Navigator browser, meaning that web pages didn't always render the same way on both. And importantly, web applications using JavaScript rarely behaved the same way on both. (To be fair, Netscape didn't make any effort to be compatible with Internet Explorer either, an act of hubris that cost them dearly.)

As Internet Explorer's market share increased, the incompatibilities between the two browsers succeeded in destroying the potential value of the browser as a reliable application platform. Having been web developers ourselves during the “browser wars”, we remember our companies establishing formal policies against the use of JavaScript in web applications, because such applications could not be guaranteed to work across browsers. The policy was to perform all processing on the server side (even simple calculations) and use the browser to display no more than simple HTML pages generated by the server.

That's how the currently popular web application architecture began. In what Sun's Scott McNealy called the struggle of “Mankind versus Microsoft”, Microsoft (perhaps understandably) used its market power to prevent the development of a rival application platform layer above the level of the operating system, and Mankind retreated to the server side to try and manipulate the user interface from a “safe” vantage point. It was an architecture dictated by market reality and nothing more. Manipulating the client-side user interface from the server side is like threading a needle with buttered boxing gloves, but Microsoft's client-side monopoly left the rest of the industry no choice.

Microsoft won that round and Mankind lost. But there is no reason to continue with the architecture born out of that defeat, because the territory lost in that round has since been regained, thanks to Firefox and the WaSP (Web Standards Project). Today, it is eminently possible to treat the browser as a full-fledged application platform. Incompatibilities between different browsers are now minimal.

Our paper recognises the new reality and we don't see anything today that continues to justify an approach that drives the user interface from the server side. Move over, web frameworks.

Flaw 3: Data Interchange Restricted to Request/Response Semantics

HTTP is a request/response protocol, which means that the Business Logic tier can only respond to requests from the Presentation tier and cannot initiate any unsolicited Data Interchange operation. However, there are any number of real-world application use cases where this kind of behaviour may be required, and these could be generically categorised as "event notification". One could view the entire client application as a "view" into a "model" that is held behind the service tier. When the model changes, the view must be notified. However, current thin client technology does not support server push. In most such cases, the workaround is to have the client *poll* the server periodically, converting "push" to "pull". This is another example of a system that "sort of" works and thereby weakens the impetus for an overhaul. Having to implement server-initiated notification by client-side polling is not just an inefficiency but a symptom of an architectural limitation.

There are two reasons, however, why we don't see this flaw as requiring an urgent fix.

Although SOA thinking has progressed a great deal in the business logic tier, there is still a reactive, request/response flavour to services. Analyst group Gartner goes so far as to divide progress in this area into two distinct phases – SOA and EDA (Event-Driven Architecture). We do not agree with Gartner that the two are distinct architectures, but it is certainly true that the event notification side of the business logic tier has not yet been standardised. At the time of writing, there are two competing Web Services specifications (WS-Eventing and WS-Notification), and a clear winner is yet to emerge. Upgrading the Presentation Tier to support peer-to-peer capability before the Service Tier does will achieve nothing.

The request/response nature of HTTP has also so much influenced modern networking practice that the very infrastructure has ossified around it and cannot readily change to support peer-to-peer Data Interchange. Firewalls, NAT (Network Address Translation) and other core network capabilities are designed with the implicit assumption of the dominant application protocol being request/response. Firewalls frown on unsolicited server-to-client Data Interchange operations. One could argue that converting applications to a peer-to-peer model would open up many security issues, but here is yet another example of an implicit dependency that "SOA thinking" would eliminate. Security is an orthogonal concern that can be addressed through a suitable A&A (Authentication & Authorisation) mechanism. Embedding restrictions against peer-to-peer communication within firewall rules creates a needless barrier to peer-to-peer networks – a "network externality". But that's another aspect of current reality.

Towards a Better Architecture for Thin Clients

Web technology suffers from some fundamental flaws[1]. But all is not lost. For one, we can take responsibility for *data integrity* and Data Interchange out of HTML's irresponsible hands and entrust it to an XML-based Data Interchange mechanism instead. The most popular such mechanism is AJAX. The X in AJAX stands for XML, so AJAX can readily deal with properly defined and structured data. Being asynchronous, AJAX is also decoupled from any involvement in Presentation Flow. It is purely about Data Interchange.

Thanks to AJAX, we can design Presentation Flow independently of Data Interchange. Or can we? There is a point where the two must come together, because data from a Data Interchange operation (a GET) must be inserted into a "blank" page or template to form a properly formatted web page for the user. Where should they come together? Where the client state is maintained, of course. Since AJAX interactions are initiated on the client side, it suggests that the component that holds client state should be on the client side as well. What should this component be?

The rich client model (which doesn't suffer from the Presentation Flow/Data Interchange coupling) suggests that the MVC Controller is the component that maintains client state and that orchestrates both Presentation Flow and Data Interchange operations.

So one obvious way to correct the thin client architecture is to implement a *true* MVC framework on the client side, not a Front Controller-based framework on the web server. That means that all "Presentation Flows" must occur within the currently-loaded web page through JavaScript magic. The entire set of coherent application functions *must* reside on a single web page. If the web page is ever replaced through a fresh GET operation, the Controller is flushed away and client state is lost. So one workable model is the Single Page Application (SPA). The entire application resides on a single HTML page, perhaps with a number of hidden DIV segments. A JavaScript-based MVC Controller holds client state and may manage Presentation Flow through DHTML, by selectively hiding and showing these DIV segments. It also knows when to initiate AJAX Data Interchange requests. These requests could either map trivially to REST operations or to SOAP-based Web Services in the service tier. JavaScript libraries for XML/SOAP manipulation would be required.

A somewhat less elegant alternative is for the MVC Controller to reside within a hidden "controller" frame on the client side that is never replaced when new pages are loaded into an "application" frame. With this architecture, Application Download can even take place piecemeal, providing the "small footprint/no offline" feature of a thin client application. A web server passively serves up page templates in response to GET requests from the MVC Controller. The Controller then populates these templates with data that is returned by various Data Interchange requests and displays these pages within the application frame. (The web server does not drive Presentation Flow in this model either.)

1 There is nothing wrong with HTTP as a protocol, but the "Web 1.0" model of HTML-over-HTTP is broken, for the reasons covered above.

Both these models are eminently implementable using AJAX, and at last we see light at the end of the decade-long tunnel of "Web 1.0" technology. We are reminded of the quote from Henry VIII:

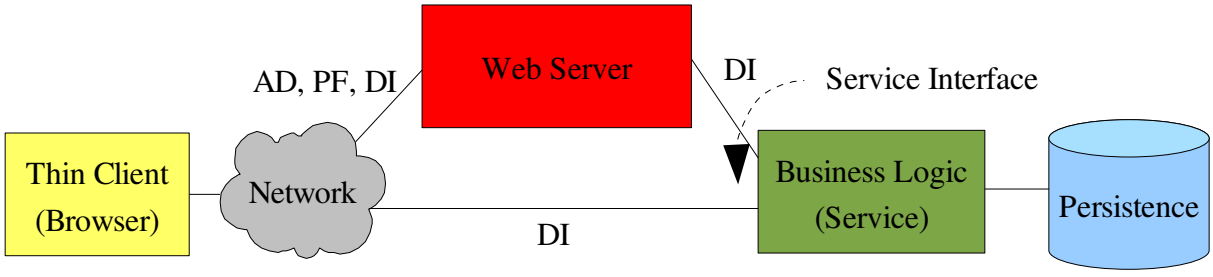
> *Had I but served my God with half the zeal I served my king, he would not in mine age have left me naked to mine enemies.*

Indeed. Had we but invested in client-side JavaScript half the resources we invested in server-side Java, we could have had AJAX far earlier (and we could have avoided wasting our time on fifty different Front Controller frameworks).

In sum, we like three things about AJAX (quite apart from its eye-candy, which we like too). What we like as architects has to do with what AJAX stands for – Asynchronous JavaScript and XML:

1. It's Asynchronous. That means it's designed to decouple Data Interchange from Presentation Flow.
2. It emphasises JavaScript-based intelligence. That allows us to move responsibility for the user interface back from the server to the client. It's possible to think of an MVC Controller in JavaScript that manages client state and orchestrates both Presentation Flow and Data Interchange.
3. It uses XML. That means it respects data. More importantly, in those Data Interchange operations that occur across the Service Interface, the same XML documents could be used without translation.

However, raw AJAX is just a capability, a bundle of potential in need of a governing architectural model. The way AJAX is used today merely complicates the standard web model. The Thin Client can now call Business Logic directly, *in addition to* the Web Server doing so. This doesn't do us any favours. It just means the old model of the web server driving Presentation Flow and mediating Data Interchange may continue, with AJAX interactions just hanging off to the side, so to speak.



Note that it is entirely upto the application developer how the Data Interchange function is split between browser and web server. So while AJAX delivers some exciting new capabilities, it raises architectural questions that it itself doesn't answer.

The other subtle aspect of raw AJAX is that it doesn't *mandate* the use of XML as the format for Data Interchange. JavaScript Object Notation (JSON) is also supported by AJAX frameworks. In fact, JSON is actually preferred by many developers over XML because it's much easier to use.

It falls to us to strike a dissenting note here too. JSON does not help us in our goal of achieving better integration with a service-oriented Business Logic Tier. The following table compares raw HTML-over-HTTP, JSON and XML.

	HTML-over-HTTP	**JSON**	**XML**
Data Structures	Flat set of name-value pairs	Hierarchical	Hierarchical
Data Types	"String" only	Loose typing with JavaScript rules	As defined in schema
Data Constraints	None	None	As defined in schema

It is only XML that can comprehensively enforce data integrity. Besides, the Service Tier already uses XML as its data format of choice. The Presentation Tier must choose a compatible data format that does not weaken the chain of data integrity of the application as a whole. Regrettably, we cannot endorse JSON as a suitable data format for Data Interchange.

XML transmission and processing overheads used to be onerous, but with significant advances in these areas (e.g., MTOM and StAX) and given inexorably improving hardware horsepower and network bandwidth, we have no hesitation in making XML a core part of our model.

The bulk of Part I of this paper has been spent on discussing the drawbacks of the thin client model. Note that while AJAX is a common Data Interchange mechanism, it is not the only one that can be used in the Presentation Tier, merely the best-known and best-supported thin-client Data Interchange technology that satisfies SOA requirements.

Rich clients do not suffer from the problems we described to the same extent. The model we propose therefore suits rich clients with far less modification.

The second part of this paper will attempt to describe a consistent end-to-end architecture for an application that covers the gamut of labels such as "thin client", "rich client", "smart client", "AJAX", "Rich Internet Application", "SOAP-enabled", "RESTful", etc.

In the third part, we list some popular Presentation Tier technologies. In a later paper, we will compare these technologies against our architectural model and critique them.

Part II – The Way Things Should Be

The SOFEA Model

We propose a model that addresses what we believe to be the three flaws in current client technology. When these are fixed, there automatically arises a unified model for clients (the labels of "thin" and "rich" become moot) and integration with the Service Tier becomes seamless.

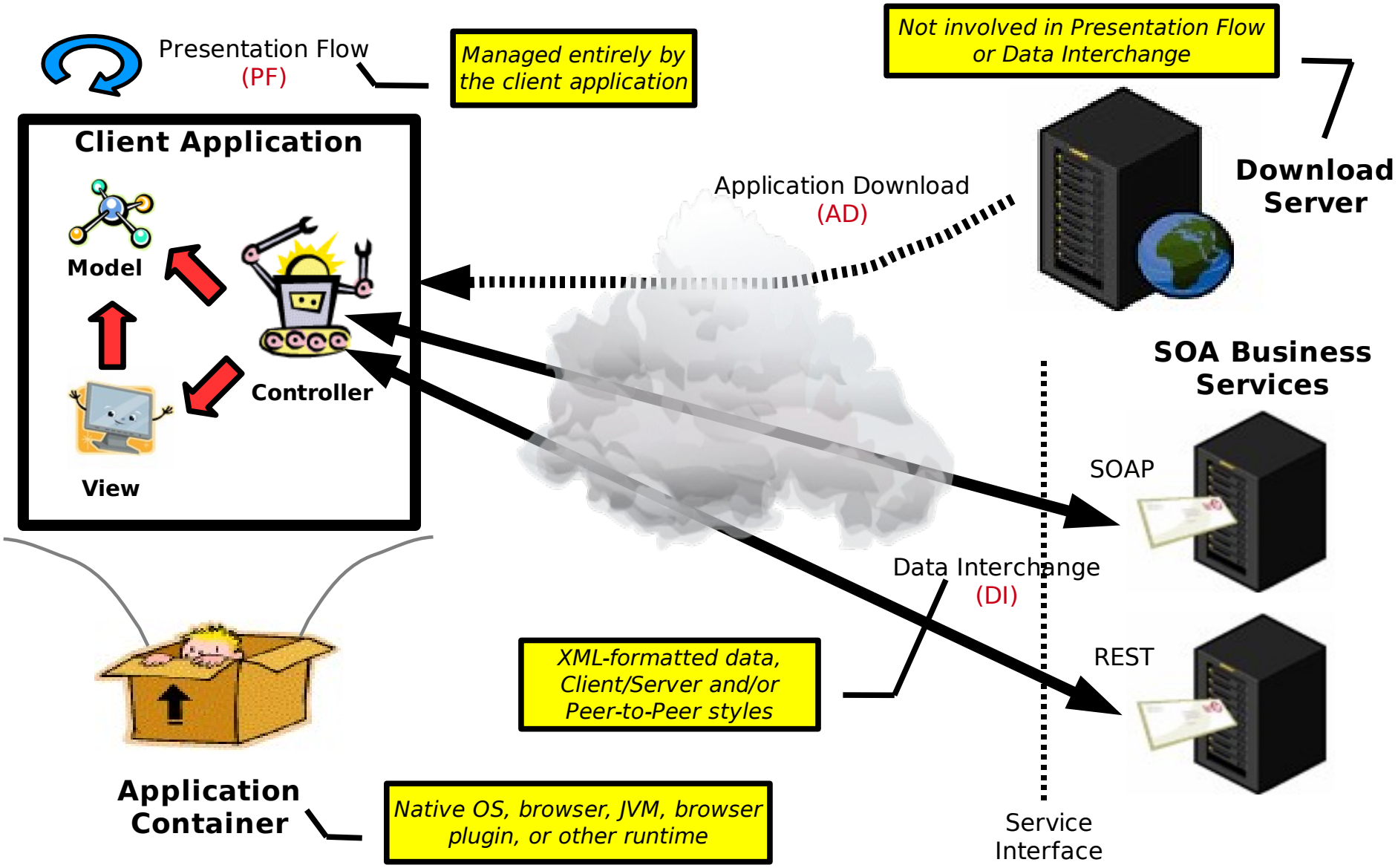
The defining principles of our model are:

0. Decouple the three orthogonal Presentation Tier processes of Application Download, Presentation Flow and Data Interchange. This is the foundational principle of our model, which is why it is numbered zero.
1. Explore various Application Download options to exploit usefully contrary trade-offs around client footprint, startup time, offline capability and a number of security-related parameters. This will allow the application to adopt the characteristics of "thin" or "rich" clients as required.
2. Presentation Flow must be managed within the client. If a server-side component is to be used at all, it must be something simple such as a collection of page templates. Under no circumstances must Presentation Flow be driven by a server-side component.
3. Use XML to define data structures, data types and data constraints for the application end-to-end. Try and align the XML schemas used by the Presentation Tier to those defined by the Service Interface to make the integration seamless. Ideally, try to use a peer-to-peer Data Interchange model between the two tiers rather than the client/server model. Given the impediments to enabling peer-to-peer capability in the Service Tier as well as in the network infrastructure, this may be considered the optional part of our model for the immediate future.
4. We are not overly prescriptive about this, but we recommend the use of the Model-View-Controller (MVC) pattern to build the Presentation Tier. The MVC Controller will manage client state and drive both Presentation Flow and Data Interchange processes. The DHTML/AJAX combination is one proven technology solution that could be used in browser-based applications.

We call this model SOFEA (pronounced Sophia), or Service-Oriented Front-End Architecture.

Diagrammatically, the model looks like this:

SOFEA (Service-Oriented Front-End Architecture)



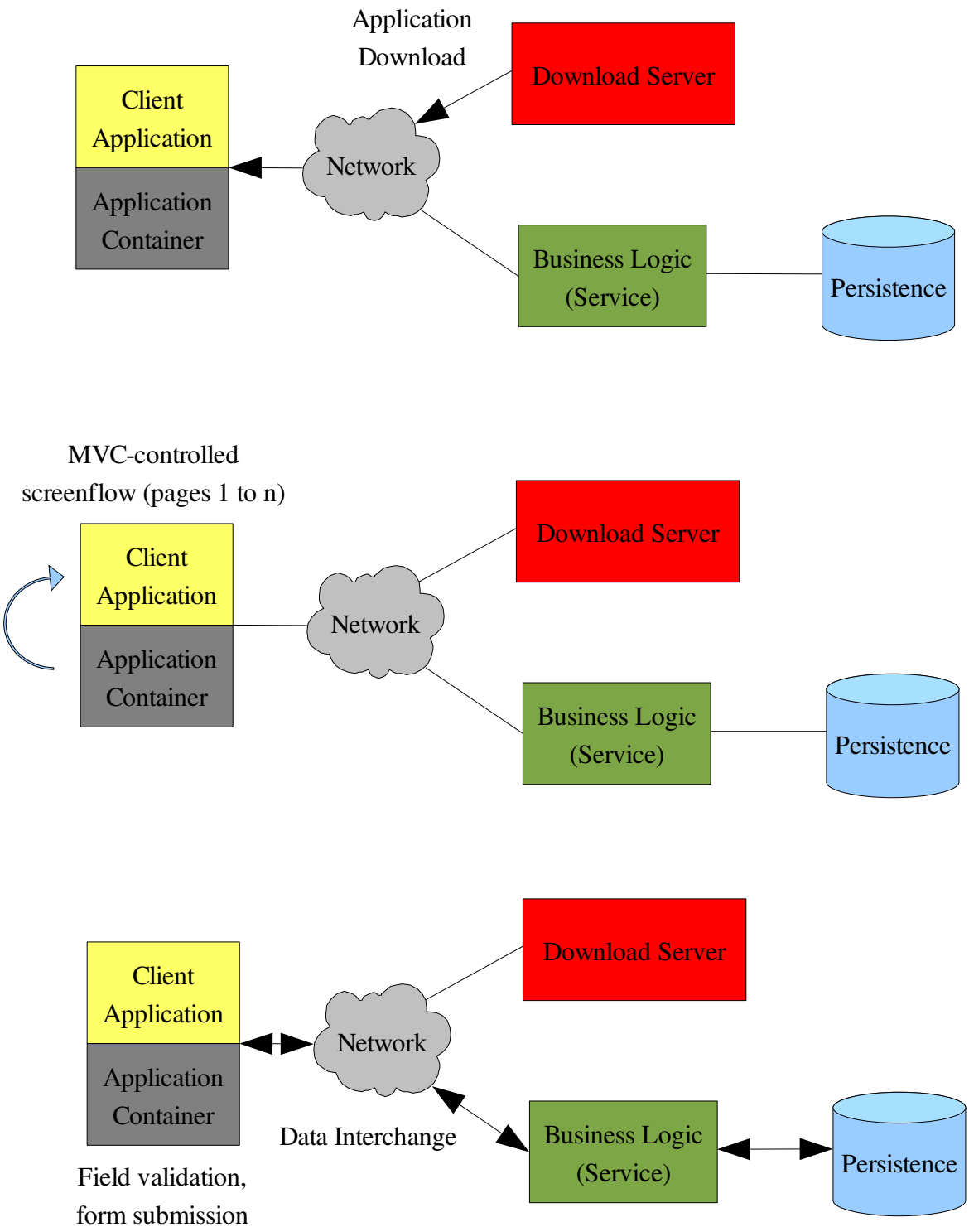
At first glance, this doesn't appear to be very much different from the AJAX model as used today, but there are some crucial differences:

1. The Download Server never invokes Business Logic. It is emphatically not a "tier" between the Client and Business Logic, i.e., our architecture is strictly 3-tier, not N-tier.
2. This model is not restricted to "Thin Client" architectures. "Rich" and "Smart" clients map to it as well. The "Application Container" is a generic concept that corresponds to a browser in the Thin Client model but to other technologies in a Rich Client model, e.g., JVMs and Flash players.
3. Regardless of implementation technology, the entire client application is built using a *standard* MVC architecture, not the pseudo-MVC Front Controller pattern common to web frameworks.
4. Not being tied to HTTP, the model is open to peer-to-peer Data Interchange.

Note that Application Download, like the Application Container, is also a generic concept. In Rich Client architectures, Application Download takes place *all at once*, and entirely independently of application execution. For example, we may download and install an executable from SourceForge (our Deployment Server) onto our workstation but only use it for the first time after a couple of weeks. With Thin Client architectures, Application Download could still take place page by page concurrently with application execution, provided only page templates are used on the server side with no presumption of client state management. Hybrid approaches are possible, such as the Java WebStart/JNLP model which can download incremental updates to applications. As we saw, differences in Application Download can impact capabilities such as small footprint/quick startup versus offline capability/disconnected mode operation. The "Deployment Server" is usually a Web Server in either model, serving up either web pages or entire applications.

Example – A Multi-Page Form Conforming to SOFEA Principles

Here's how a multi-page form would be implemented conforming to the SOFEA model, without pre-supposing either a rich client or a thin client paradigm.



This is very different to the traditional web application (even most AJAX ones), because each page in

a multi-page form has different elements, and web developers tend to devote a physically distinct page to each, which is then served up from the web server using the Front Controller pattern as the Presentation Flow progresses. The use of AJAX is typically timid and works around this basic structural flow (e.g., per-field validation, pre-population of fields, context-sensitive dropdowns, etc.).

In a thin client implementation of the SOFEA model however, the entire form is expected to be contained within a *single* physical page which manages the flow between logical form "pages" through an MVC approach (e.g., a JavaScript-based Controller may selectively hide and show form DIVs within the page). The client Application is therefore responsible for its own Presentation Flow and associated state. (Server-side page templates could, of course, be used.)

There could be Data Interchange events that occur during the course of the Presentation Flow. Far from breaking the model, it illustrates that the two work together seamlessly. The MVC Controller initiates Data Interchange operations and updates the MVC Model along the way. Field validations, pre-population and context-sensitive changes to the form are all examples of how Data Interchange can be interleaved with Presentation Flow. Throughout this process, the Deployment Server is not engaged at all. In a traditional web application, the web server would be needlessly and continuously in the loop.

That's how an application of moderate complexity is modelled using SOFEA.

How would the SOFEA model handle a new requirement to add "save and resume" functionality to a multi-page form, where the user needs an option to save a *partially-completed* form and resume it in a later session?

Well, apart from changes to services to accept partial data, the Application simply provides an option to initiate a new Data Interchange at intermediate points during the Presentation Flow. The MVC Controller is the only component (above the service tier) that is affected by this change. The Deployment Server remains unaffected in this changed operation at run-time, as it should be. (Think about how this requirement would impact the design of a traditional N-tier web application.)

Part III – The Way Things Are Shaping Up To Be

These are some of the most popular Presentation Tier technologies available today:

1. DHTML/AJAX frameworks for Current Browsers
    - Largely handcoded with third party JavaScript libraries
    - Google Web Toolkit (GWT, GWT-Ext)
    - TIBCO General Interface Builder
2. XML Dialects for Advanced Browsers
    - XForms and XHTML 2.0
    - Mozilla XUL
    - Microsoft SilverLight/XAML
3. Java frameworks
    - Java WebStart (with/without Spring Rich Client)
    - JavaFX
4. Adobe Flash-based frameworks
    - Adobe Flex
    - OpenLaszlo

In a future paper, we hope to analyse and critique each of these technologies from the SOFEA standpoint, and also provide guidelines on how best to use it in accordance with SOFEA principles. We think that would be of considerable practical value to developers.

Conclusion

Although it seems presumptuous on our part to claim that we have "solved" the end-to-end integration problem, what is probably true is that recent paradigms and technology breakthroughs have brought the SOFEA model closer to conceptualisation, and someone or the other was bound to suggest it. It happened to be us.

The contributions of the SOFEA model are the following:

1. A cleaner architectural model for the Presentation Tier that decouples the orthogonal concerns of Application Download, Presentation Flow and Data Interchange.
2. Positioning of the web server as a Download Server alone. The evils of web server involvement in Presentation Flow and Data Interchange are avoided.
3. Affirmation of MVC rather than Front Controller as the natural pattern to control Presentation Flow.
4. Assurance of end-to-end data integrity in Data Interchange, which traditional thin-client technology does not and cannot enforce.
5. Unification of the thin-client and rich-client models, now seen as an artificial distinction.
6. Support for SOAP- and REST-based business services, and a natural integration point between the Presentation and Service Tiers.

Finally, every good framework needs a cool logo, so we propose the following for SOFEA:

SOFEA Logo

Logo to be displayed by applications conforming to SOFEA principles

The black stroke represents Application Download, the blue and red arcs stand for Presentation Flow, and the green stroke represents Data Interchange. The overall logo resembles the Greek letter Phi, which represents the fricative consonant 'f' in SOFEA. That should be abstruse enough to be cool.

We believe SOFEA application development will become mainstream with a wide variety of implementations in less than a couple of years, because it just makes sense, and the constraints that necessitated a departure from these principles in the first place no longer hold good.

And to those building the next great server-side presentation framework, we have a word of advice – don't.

About the Authors

(l to r: Ganesh Prasad, Rajat Taneja, Vikrant Todankar)

Ganesh Prasad (g.c.prasad@gmail.com, http://wisdomofganesh.blogspot.com) is a Senior Architect (Presentation, Integration and Java Technologies) within the Enterprise Services division of Westpac Banking Corporation, Sydney, Australia. He has 20 years of IT experience covering software development, design and architecture.

Rajat Taneja (rajattaneja@optusnet.com.au) is Chief Architect, Zurich Financial Services, Australia.

Vikrant Todankar (vikrant.todankar@gmail.com) has many years of experience designing and building J2EE applications in the Financial Services sector. More recently he has been working on application integration solutions and consulting for Java web applications. Vikrant has been a senior consultant with EDS Australia and is currently with Hyro Limited in Sydney.

The views expressed in this paper are the personal views of the authors and do not necessarily represent the views of their employers.

Rhys Frederick reviewed this paper. Rhys is a Principal Consultant with Object Consulting, Sydney.