A Unified Mobile Architecture for the Modern Data Center

The issue with the predominant multi-tier architecture is that it is designed with a browser in mind. Service/API Gateway mobile middleware bridges the gap between legacy apps and the sea of heterogeneous mobile platforms, operating systems, and programming languages used today.

Abstract
This paper is geared toward the CSO, CISO, application architect, information security manager, developer or network administrator that takes part in the development and deployment of the supporting architecture for the mobile application economy. We look at the nature of mobile applications and then make recommendations on the essential server side components required in this new reality, especially as they contrast the well accepted world of web applications. This paper posits that mobile applications, especially native apps, are quite different than the client/server world we are familiar with, even though they happen to use HTTP for communication.
Introduction
Most organizations today concentrate on how to fit mobility and the ‘app’ economy into existing infrastructure with minimal changes. Enterprises are still learning how to change their server-side and middleware architecture to fit the newer, faster, quicker, more efficient mobile taxonomy.

It would be difficult to argue that there is something hotter than mobile apps right now, as we are even starting to see Enterprises deploy mobile applications for multiple stakeholders including customers, partners, and employees. Even the president is in on the action; in May 2012 President Obama sent a memo to the heads of every major Federal agency instructing them to make mobile apps for each department available to U.S. citizens by the end of 2013.

According to Forrester research, many of these applications have been productivity applications such as people-finder applications and conference room booking applications, called “B2E”, or Business-to-Employee. Some larger retailers and service providers such as Home Depot, Amazon, and eBay have apps too. Some of these apps compete with traffic garnered via corporate websites by essentially offering the same or similar experience available in a browser, whereas other apps augment existing business processes.

As apps continue to proliferate with the advent of Windows 8 and the growth of Android, iPhone and Windows Phone platforms, we ask the question: What will the enterprise server side architecture look like in this new reality? What essential changes are driven by application requirements that weren’t required in the browser-only world?

Further, how does this architecture intersect with enterprise security, business scalability, and performance requirements? What changes should the Enterprise prepare for? Where should architects and technical leaders look when designing their next generation architecture for mobile applications?
The Status Quo
To grasp our questions, we can contrast them with a paradigm that has wide acceptance in the proverbial “field”. The modern SaaS application or classic multi-tier application architecture for web applications is often described in three tiers:

1. A persistence tier – for holding user and application data
2. An application server – for business logic and controlling views, generally through a model-view-controller (MVC) paradigm.
3. A web server – for rendering views to browsers

All of the previously listed above can be separately scaled and implemented with various technologies. For example, the application server may be Java, Python, Microsoft .NET, Ruby on Rails, C++ applications, XSLT centric applications, or any of a wide variety of frameworks.

The persistence tier might be SQL based, or a high-performance NoSQL database, such as HBase, and the web server is generally Apache but could be IIS, Nettly or Tomcat. It may also include systems beyond pure databases such as BI (Business Intelligence), BA (Business Analytics), data warehousing, and mainframe systems.

In the current architecture, the client is almost always a web browser and the security model is HTTP-centric, depending mostly on SSL. Identity management is usually LDAP or a legacy web access management product. For newer applications where user data is shared between parties, OAuth is also becoming prevalent for authorization as well.

All of the technologies in this multi-tier architecture are location independent. For instance, the persistence tier and application server may be running in the cloud, and not "on-premise." An entirely different argument could be made that the multi-tier architecture need not even be owned by the Enterprise in a pure public cloud model.

The trouble is, we are not entirely there yet. Many large, vertically integrated Enterprises gain specific business advantages when running their own infrastructure. That is, large enterprises may use the cloud for bursting and auxiliary functions, but many aren’t yet ready to transition their entire core applications off-premise for business, legal, risk and other reasons that aren’t pure cost or technology decisions. A hybrid cloud model is predominant in most large Enterprises today and is the basis for the unified architecture we are proposing here.

The API Economy
Of key importance in the current architecture are API calls. If we consider RESTful APIs, these can be exposed directly from the application server and should implement API-centric protection mechanisms in the application server itself, or be decoupled to a security gateway. For scalability, we can even scale each of these independently on different cloud providers if the need presents itself. As a side note, RESTful interfaces are overtaking web traffic for the largest SaaS applications. As of mid 2012, more than 60% of all logins to the popular Salesforce.com platform are through RESTful APIs, and 40% are through traditional browsers. RESTful API calls, which have a strong relation to mobile applications, are overtaking the web for the largest application providers.

The issue with the predominant multi-tier architecture is that it is designed with a browser in mind. What would this standard architecture look like if we served mostly apps instead of browsers?

If the Salesforce trend continues and more logins are through applications, what happens to our existing multi-tier architecture? To answer this question, we need to investigate the mobile application landscape and extract the common characteristics of the modern mobile application. Only then can we really decide what common functionality is required on the server side.

---

1Data taken from a Salesforce.com webinar, delivered in June 2012 entitled “Manage Identity and Access to All Your External SaaS Apps from Salesforce”
Mobile Applications

First, we have to distinguish between two main categories of mobile applications. These include native applications and mobile web applications. There is also a third category, a wrapped mobile web application, also called a hybrid web application, which is a web application wrapped in a native operating system “viewer” designed for the mobile O/S itself.

Native applications are compiled for a specific device type and operating system revision, and depending on the operating system, such as Android, different SDK revisions are needed for different operating system versions. In the majority of cases the development environment is provided by the phone and platform manufacturer.

For instance, one iPhone the programmer writes in Objective-C, on Android a Java SDK is used, and on Windows 7, the programmer writes in C# and uses Visual Studio. Some manufacturers, such as Research in Motion (RIM), the makers of Blackberry, support development environments in multiple languages, for instance Java and C/C++.

This type of native application is almost a special breed. It is a networked application that will end up doing most of its communication over HTTP – but it’s not a browser.

We might say that native applications can achieve better performance levels as they have access to all of the platform features, but they have low portability.

Imagine writing an application in C# for Windows 7 and then porting it to the iPhone. This is almost like writing a second application from scratch. The UI controls are different, the platform layer is different, and the persistence layer is different. Further, native apps have entirely different features compared to what you’ll find in a web application - take push notifications for example. This is a mechanism for a native application to receive information and updates, even when the app itself is closed. Browser based apps don’t even have this concept.

Native mobile applications are network based and illustrate that a browser is not required for a network based HTTP application. We always knew this, but the proliferation of mobile applications has made this fact more obvious. To put it simply, native mobile applications don’t really fit the client/server model of the web, even if they happen to use HTTP for communication.

Native mobile applications have more in common with desktop (gasp) or “thick client” applications with on-demand network I/O that use synchronous HTTP by happenstance and convenience as well as non-HTTP communication. We might even draw a closer parallel between Java Applets, which went out of style about 5-10 years ago and native mobile applications.

Today’s native mobile applications are rich. They perform a good deal of computing on the device, they query local databases, they have complex-event driven UIs, they support 3D graphics engines, they make on-demand network access to personalized data, and they perform real-time analytics with this data. Because of this, enterprises should be building applications that allow for the server-side processing of any of those components anywhere as necessary, for example some processing in the device itself, some in the enterprise and some in a cloud based processing model. The best native mobile applications will be composite distributed applications with “split processing”, some on the device, some in the Enterprise datacenter, and some by a cloud service.

Moreover, modern mobile applications live in a sea of heterogeneous mobile platforms and operating systems, each with their own programming language and best practices. To paraphrase Ian Finley from Gartner, client side mobile development is “the wild wild west” in terms of programming languages, operating systems, devices, and capabilities.

For Enterprises, this means breaking out of the controlled “client/server” mindset. With mobile applications, variability of the client is the rule rather than the exception. From the Enterprise point of view, the only real common denominator is that mobile apps tend to communicate with the Enterprise using HTTP, but even this is not a hard-and-fast rule.

The experience on a mobile application is user-centric, intuitive, and feels much more like a consumer play-toy compared to an Enterprise productivity application. In this sense, mobile apps are thick, monolithic applications that have no obligation to exclusively obey the brittle web-only “Request” / “Response” paradigm that has gripped the Enterprise for the past two decades.
In the past, it was ubiquity that drove browser adoption, but now it seems ubiquity is here, but in a different form, provided by the application marketplace. Even platforms like the Windows Phone, which has dismal adoption compared to Android or iPhone provides broad support for apps. The trouble with native apps is that they require programmers, and programmers are expensive, especially when each native platform requires specialized platform knowledge. To address this, companies such as Antenna, Appcelerator, and SAP/Syclo have developed cross-platform development environments that allow an Enterprise to author mobile applications in an Eclipse or other similar standardized development environment and cross-compile them for all popular platforms. These development environments typically lower the bar to entry, for example allowing the developer to write in JavaScript and end-up with a shiny, native iOS or Blackberry application.

**Security and HTTP I/O**

While the basic I/O model for native applications is still HTTP, native mobile applications are thread optimized for an optimal user experience. In particular, synchronous connections are confined to a separate thread and then used to update the application state and user interface. This gives a smooth user experience in the face of blocking I/O calls.

For native applications, the strength of the security controls lies in the ability of the application developer to write secure code, but for the most part, the developer gets to use HTTP for a good deal of communication which means transport level security will likely be implemented correctly as a function of the SDK, but trickier protocols such as OAuth are left for the application developer to figure out.

**Mobile Web Applications**

Mobile web applications, on the other hand, run inside the web browser on the device, built on an abstraction layer that is one level removed from the native phone capabilities.

To date, JavaScript frameworks such as jQuery Mobile and Sencha have become very popular. These frameworks rely on web standards such as HTML5, CSS3 and JavaScript, lowering the bar for both the authoring of applications and their distribution.

These frameworks have advanced user interface capabilities but because they run inside the mobile phone’s browser, they may not have access to all of the phone’s native features. Some examples here include robust persistent storage, disconnected operation, push-notification support, native location capabilities, such as GPS or location-awareness, asynchronous raw TCP communication, and OTA (Over-The-Air) software updates.
Further, they may not perform as well for certain types of applications that require persistent socket connections, though the HTML5 WebSocket API promises to fix this. The main advantage of a mobile web application is portability and reduced skill-set requirements, but this also comes at a price as not all phones have the same level of JavaScript or HTML5 support. Also, to date, these frameworks suffer from the cross-domain browser problem, while ‘solved’ with JSONP, is not considered a secure means of a cross-domain call as it introduces the opportunity for code injection, and other browser-based exploits. Further, it doesn’t fundamentally solve browser compatibility or outdated browser version issues. There have been some additional W3C efforts, including Cross-Origin Resource Sharing (CORS) that may improve this in the future, but at the time of this writing, this spec was only in working draft stage.

We can summarize the salient differences between these two predominant flavors of mobile applications in the following table:

<table>
<thead>
<tr>
<th>APPLICATION TYPE</th>
<th>Native (Runs on phone operating system directly)</th>
<th>Web Mobile (Runs in browser or O/S browser container)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLICATION</td>
<td>Faster; code is closer to the phone operating system</td>
<td>Slower; code is written on an abstraction layer of web technologies (JavaScript, CSS, HTML5)</td>
</tr>
<tr>
<td>PORTABILITY</td>
<td>Low – application must be ported across languages and environments in most cases. Cross-platform tools can reduce development costs</td>
<td>High - but depends on JavaScript support and vulnerable to HTML5 fragmentation</td>
</tr>
<tr>
<td>APPLICATION CHARACTERISTICS</td>
<td>Mostly Monolithic – distributed through an app-store. Application logic and persistence are managed with interwoven components. Best support for offline features, native data storage and push-notifications. Networking is the exception – HTTP can be used for data retrieval. Updates require re-download and reinstallation.</td>
<td>Web-Oriented – distributed through a URL. Application logic is pushed down from the server in JavaScript, HTML5/CSS. Limited offline mode support, data storage and Application can be updated more easily by serving updated pages and JavaScript.</td>
</tr>
<tr>
<td>NETWORK I/O</td>
<td>Synchronous HTTP confined to a separate thread. Use of persistent socket connections for asynchronous notifications and altering. Supports over the air (OTA) capabilities for application updates.</td>
<td>Synchronous HTTP or HTML5 Websockets (Emerging)</td>
</tr>
<tr>
<td>API CALLS</td>
<td>Handled through native HTTP library or raw TCP</td>
<td>Handled through AJAX calls; JSONP required for cross domain API calls</td>
</tr>
<tr>
<td>CLIENT SECURITY POSTURE</td>
<td>Phone has access to the file-system to read and write files. Data read from remote sockets must be parsed. Security beyond HTTPS requires a toolkit or custom code. More susceptible to malware – native code is closer to the O/S.</td>
<td>Basic security options are confined to what is provided by HTTPS. Additional protocols such as OAuth require a toolkit. Browser or O/S wrapper provides a sandbox.</td>
</tr>
<tr>
<td>SERVER ARCHITECTURE</td>
<td>Not well-defined – a mix of custom middleware and software. Mobile middleware is often segregated to its own ESB.</td>
<td>Similar to web server / app-server model with content optimization.</td>
</tr>
</tbody>
</table>
Native applications are the crème of the crop. Higher development costs aside, a native application will always be able to provide more functionality when compared to the same application confined to a browser or browser-container. This is true because the programmer has access to richer platform resources through the native SDK. Code runs faster, and these apps have more control over threading, UI controls, networking protocols, access to external cloud providers, physical device features (cameras, GPS), 3D graphics, and on-board persistent storage. Browser-based apps have always promised parity with desktop apps, but they never quite get there. The situation will be largely the same on mobile devices. Over the long run, web mobile applications based on JavaScript/HTML5 will approach parity, but native mobile applications will provide a snappier user experience and richer functionality set.

Native functionality, however, must be traded off against time to market, ease of updates, and functional requirements. If this is true, what common characteristics might we associate with the best native mobile applications, and further, how do these characteristics translate to server side or mobile-middleware capabilities, especially those that focus on Enterprise use cases?

1. **Enterprise Identity and Single Sign-On:** In social consumer applications user identity is held by one or more identity providers such as Facebook or Google, and identities are connected to a networked graph of friends or acquaintances. In the Enterprise, the user is identified by an Enterprise identity provider and authenticated via a directory or in some cases by a database or data-store. In the context of a mobile application, enterprise apps will need to integrate with a networked graph of colleagues or internal Enterprise social network. Also, Enterprise applications will require federation, especially in the case of web mobile applications or hybrid applications, but also for native mobile applications - which is a yet unsolved problem.

   This functionality implies rich server side support for both browser single-sign-on functionality and OAuth processing, with the ability to broker identity logins from mobile applications to Enterprise identity stores, such as LDAP. For Enterprises that require step-up authentication, device location and context will become important as well as strong authentication through one-time passwords or additional factors.

2. **Client Trust and API Data Security:** For Enterprise mobile applications, the client device (phone or tablet) is fundamentally non-trusted as it is not issued or owned by the enterprise. This is the essence of BYOD – Bring Your Own Device. The security posture of client devices can be bolstered with additional software or code hooks that allow mobile application management from a remote IT department, but this approach burdens the device and mobile applications themselves with coded-in security. It also raises the cost of mobile application development, as code hooks represent another SDK that must be managed along with the cacophony of devices and developer SDKs already in tow. In addition to raising the cost, it also locks in the security to a specific standard and a specific device set making the cross device portability a major issue.

   A preferred approach is to build security on the server side or on the mobile aware middleware platform, where mobile devices make their RESTful API calls. This involves user authorization with OAuth 2.0 using a public client flow and persistent data security mechanisms such as encryption, tokenization and data leak protection for data sent to potentially non-trusted devices. This data-centric approach puts a security perimeter around the data itself or data and processes/applications rather than relying wholly on coded-in mobile management SDKs.

3. **Enterprise App Store:** For Enterprises looking to offer their own private applications delivered and managed under their control, they will need to build or control their own app store. Multiple devices and operating systems can be supported, including support for both native and web mobile applications. The concept of an app store is not new from the consumer side, and the appeal to Enterprises is better management of applications and assets as well as security risk reduction, as applications are officially blessed from IT before being disseminated to client devices. Successful app stores will have to provide security controls, authorization and ubiquitous access from corporate networks and Internet locations.
App stores are generally implemented with a set of unified RESTful services and really have no analog in the browser only world, where the web server/app server played the dual role of application and application-dissemination point. Another point here is that we can expect Enterprise app stores will follow suit with iTunes, where more than just applications can be downloaded.

Enterprise app stores will also become a distribution point for other mobile-ready content such as video, audio, books and other content, but with an Enterprise bent. For example, we can imagine corporate training videos or related books, articles and communication to come from the Enterprise app store. The Enterprise App Store is also the place to control push notifications, alerts and OTA (Over-the-Air) updates. It also requires rich access control features to handle controls on who can publish new applications to the app store and handles active synchronization. For Enterprises that allow any external developer to publish a new application, virus scanning and malware controls are a must. Enterprises can also outsource this function, but in doing so may give up vertical integration and control to a third-party.

4. Activity Stream: The Activity Stream or micro feeds, popularized by Facebook News, is a common feature in consumer mobile applications. Activity streams are a feed of actions, typically represented in JSON, performed by users in the same social network or related social network. In the Enterprise, the analog is Salesforce Chatter and we can expect this type of feature to make its way to Enterprise mobile applications. For the Enterprise, activity streams will be built around the Enterprise organizational structure rather than a public social network and may also involve hybridized sources. For example, stream data will also involve service provider sources such as discover.org (for enterprise IT data) and D&B (Dun and Bradstreet) over RESTful API calls. When implemented at scale, activity streams are implemented in one of two approaches: 1) A non-relational data model that allows extensibility for user actions coupled with an efficient way to reduce a large data set to a personalized feed, or 2) A distributed queuing system. In either case, activity streams data is ultimately consumed by a mobile application over a RESTful API. On the server side, this implies either HBase or JMS with an API endpoint. Enterprises will also invariably need access control and data filtering of activity streams based on Enterprise security policies.

5. Scalable RESTful APIs: If mobile devices and their development environments are essentially disparate, the one common theme is they share RESTful HTTP protocol invocations for data access. RESTful API calls with JSON payloads have become the de facto standard to bring data to a mobile application. JSON is efficient in size compared to XML, and all web mobile applications support HTTP and HTTPS. Some application server frameworks, such as Ruby On Rails, excel at making web applications and API access available from a single code base with a configuration change.

In other cases, such as Java or PHP, API access must be specifically designed and coded in. In all cases, however, the Enterprise must worry about the scalability of HTTP for potentially millions of devices making HTTP calls into their infrastructure. One approach is to outsource this problem to a cloud provider, but another approach is to host the infrastructure in-house.

For Enterprises looking for scalable HTTP performance where they own the server side, components such as Netty or possibly the emerging Node.js offer interesting scalability options that go beyond the performance of Apache or Tomcat. Frameworks such as Node.js have additional advantages for mobile as they provide native JSON and JSON schema support.

The flipside to this generous performance architecture is service level management or metering of capacity to millions of clients. This can be for business reasons or to control hyper-active clients based on their levels of activity and usage models. Raw performance is a liability without some way to spread capacity across millions of requests and prevent certain classes of requests from eating all of your capacity. Similarly, caching capabilities for RESTful responses are also valuable, especially if multiple client mobile applications are making repeated calls for relatively static JSON responses.
6. Shared Application State: The best mobile applications share state and context between tablet, desktop and web variations. Consider an employee expense reporting application. If an expense report starts out on the desktop and then moves to a tablet or phone, the application state needs to travel between these devices. Again, the Enterprise may outsource this capability, but it may also bring it in house and offer state sharing backed by a database exposed through a secure, RESTful API.

Considering the previous capabilities, we might consider the following server-side capabilities with mobile applications in mind:

1. A highly scalable application server farm, optimized for JSON output and capable of binary asynchronous messaging for push-notifications, alerting and over the air updates (OTA)

2. A web server farm, tuned for mobile applications in general, and tuned per device brand. Graphics, images, scripts, and data must be pre-optimized for a specific device for the best performance on that device.

3. An API security gateway, for exposing RESTful services to mobile devices, both for the applications themselves and supporting an Enterprise app-store. The security gateway also handles mobile-centric identity processing, such as OAuth and API Keys, provides API threat functions and message security capabilities. An Enterprise identity store, which holds actual Enterprise identities.

4. A scalable, persistence layer for supporting activity streams and fast changing content.

5. A repository for storing application binaries and other app-store accessible content such as audio and video.

If we were to lay these items out in a unified architecture, we might have something like the following:

Diagram 1: Unified Mobile Architecture
In Diagram 2, the standard three tier architecture has grown significantly. The holistic architecture shown in the previous diagram contrasts current trends with mobile application development, which treats mobile apps as either as pet projects, owned by a single department or segregated to a standard Enterprise Service Bus (ESB), which lacks the security and scale for mobile applications.

In the unified architecture, we assume the majority of communication is initiated from native apps over RESTful HTTP interfaces. In this case, an API security gateway acts as the main front-end for RESTful for native mobile applications. By contrast, if the application is a web mobile application, a tuned web server farm serves as the main entry point. Also, as Enterprises grow, they will need to act as an integration broker, to use a Gartner term. In this model, they also require an API security gateway acting as a client to pull data to and from external cloud service providers, such as Platform as a Service (PaaS) APIs.

A lightweight HTTP/JSON optimized web server can be used to send JSON responses to both the API gateway and the web server. For data transformation, large Enterprises may also have legacy data from packaged applications, data warehouse systems or mainframes that would benefit from being made available on mobile devices, which would involve transforming this data from proprietary formats to JSON or ‘minified’ HTML/XHTML.

The App server will also need a component for binary asynchronous messaging, either directly to devices (not shown) or through a service provider (shown). Examples of service providers here include Apple Push Notification Service (APNS), Android C2DM, or a third party service.

For persistence, a scalable “NoSQL” data-store can be used for content with a fast changing data model and a standard relational database can be used to hold applications binaries and related content, compiled for different devices, which applies if the Enterprise is building their

### Table 2: Architecture Component Definitions

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>Purpose</th>
<th>Related Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>API SECURITY GATEWAY</td>
<td>Securely exposes unified RESTful APIs. Provides threat protection and trust enforcement</td>
<td>May also integrate with authorization servers, anti-malware and data-loss prevention (DLP) systems to enforce enhanced security checks on RESTful traffic.</td>
</tr>
<tr>
<td>ENTERPRISE DIRECTORY</td>
<td>Holds and manages Enterprise Identities</td>
<td>Used as a policy decision point (PDP) for the API security gateway</td>
</tr>
<tr>
<td>(LDAP or Identity and Access Management System)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FEDERATION SERVER</td>
<td>Generates assertions for single-sign-on</td>
<td>Communicates with the enterprise directory and API security gateway</td>
</tr>
<tr>
<td>MOBILE-TUNED WEB SERVER</td>
<td>Serves mobile web applications</td>
<td>May also integrate or be put in-line with a web application firewall (WAF)</td>
</tr>
<tr>
<td>LIGHTWEIGHT APP SERVER</td>
<td>Services JSON content and related business processing, suitable for mobile devices</td>
<td>Communicates with the data conversion layer and client API security gateway for securely integrating content from cloud providers</td>
</tr>
<tr>
<td>DATA CONVERSION LAYER</td>
<td>Converts legacy data from the persistence tier into JSON. Separating this component allows for increased scale</td>
<td>Communicates with the persistence tier and client API security gateway</td>
</tr>
<tr>
<td>BINARY ASYNCHRONOUS MESSAGING</td>
<td>Sends push notifications, alerts, and OTA updates</td>
<td>Communicates with the application server, generally providing push alerts triggered by application business logic</td>
</tr>
<tr>
<td>NO SQL PERSISTENCE TIER</td>
<td>Persistence for fast changing data</td>
<td>Communicates with the data conversion layer and application server</td>
</tr>
<tr>
<td>APP REPOSITORY</td>
<td>Persistence for traditional enterprise applications</td>
<td>Communicates with the data conversion layer and application server</td>
</tr>
</tbody>
</table>
own app store. A federation component can be used to return SAML assertions to web mobile applications in support of single-sign-on (SSO) and the API security gateway can enforce authentication, authorization and OAuth processing for non-trusted clients. The API security gateway can also enforce perimeter security to protect against denial of service attacks and content threats and is also used to manage service levels (SLA management) and API response caching.

We can summarize each component in more detail with Table 2 on the previous page.

An Enterprise may not need all of the previous components and in some cases, the architecture may be simpler than what is depicted in Diagram 1. However, as we can see, we are far away from our simple architecture of a web server, app server and SQL database. One way to simplify this architecture is to collapse the stack and combine functionality into a single product or component.

Intel® Expressway Service Gateway

At Intel, we believe salient aspects of this architecture, such as API security, data conversion, cloud service integration, high performance HTTP handling, SLA management, caching and on-boarding and off-boarding data to the persistence layer can be handled with a single product, the Intel® Expressway Service Gateway.

The service gateway collapses many of the components shown in the previous diagram into a single application proxy that handles all API and service security, including OAuth, secure integration and mediation to external cloud services, high performance data translation, especially for JSON content. It also handles caching of JSON responses, SLA management, and integration with Enterprise directories for policy decision support. Further, given its ability to enforce policies on any part of the message content, it can tokenize, encrypt, redact, and transform or minify data sent to mobile devices. Combined with the Enterprise Directory, application server and persistence tier, the service gateway forms the core part of the mobile middle tier. The “collapsed” architecture is shown in Diagram 2.
Important Information

Acting as a proxy, the bulk of communication from native mobile applications can be handled on the gateway itself. The service gateway, with its full capability set, combines a security gateway, high-performance HTTP engine, enterprise service bus, cloud service orchestration engine, data transformation layer and in some cases, lightweight application server into one component for the enterprise which scales on Intel® Multi-Core server hardware. In addition, it is compatibility with middleware from all major vendors and integrates with best-of-breed security solutions from McAfee for identity management and SSO, data loss prevention, and anti-malware and virus scanning, adding an additional layer of defense to API response calls.

About Intel® Expressway Service Gateway (Intel® ESG)

Intel® Expressway Service Gateway (Intel® ESG) is a software-appliance designed to securely expose or consume application services/APIs on-premise or in the cloud. It delivers cloud service brokerage capabilities including: integration, routing, data protection, middleware for legacy to mobile service enablement, and AAA security. Intel® ESG can be deployed as a soft-ware appliance, virtualized appliance in the cloud, or as a tamper resistant hardware appliance form factor with Common Criteria EAL L4+ certifications. Intel® ESG is a mature product offering that has been deployed widely by fortune 100 corporations, federal governments, and cloud providers to protect SOA, Cloud, and Hybrid service implementations. Intel® ESG has been ranked a “top developer focused product” by leading analysts and offers true developer level customization & flexibility. For more, visit: cloudsecurity.intel.com for video white board tutorials, cloud powered trial evals, and our webinar series on API Management.

More Information and Partner Program Details

Intel Application Security and Identity Products:
cloudsecurity.intel.com

Contact us by phone:
Phone: 855-229-5580

Contact us by email: asipcustomercare@intel.com

Intel® Expressway Service Gateway White Paper

Intel Corporation. All rights reserved. Intel, the Intel logo, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.