Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration

Kim J. Clark

March 18, 2015

At a high level, both SOA and web APIs seem to solve the same problem – expose business function in real-time and in a reusable way. This tutorial looks at how these initiatives are different and how they align into an evolving integration architecture. It discusses how API Management differs from the integration architectures that came before it, such as SOA and EAI.

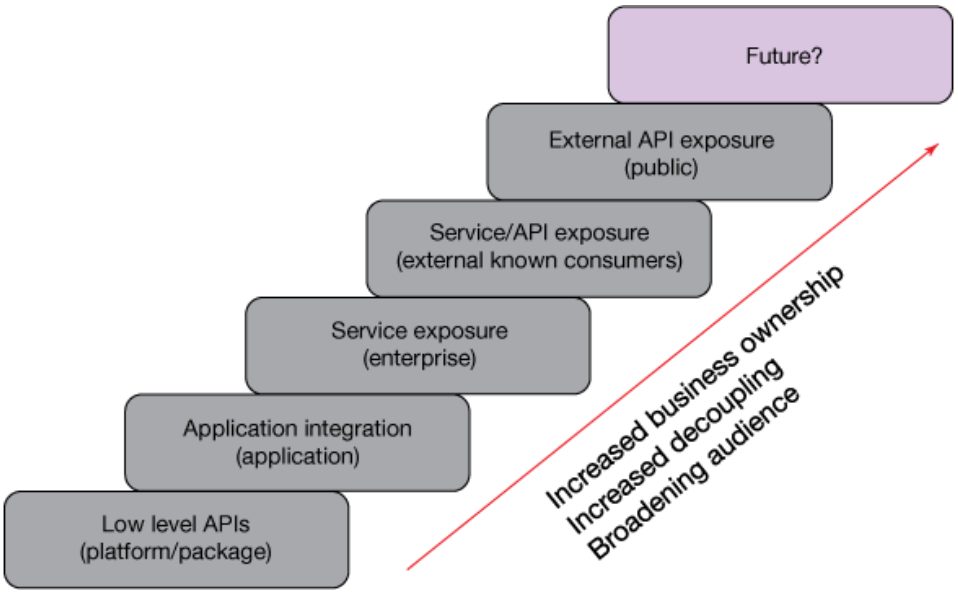
Introduction

Nearly all enterprises have multiple applications that are the system of record for their key data, and business functions upon which the enterprise is built. It is no surprise, therefore, that we see organizations wanting to progressively surface these valuable assets in these operational systems to broader audiences within the enterprise or beyond. However, it has taken time. In this tutorial, we will chart the key stages in this evolution, help you to evaluate where you are as an enterprise on that journey, and consider what actions you might want to take to mature your integration architecture towards, or indeed beyond, API exposure.

Let's take a brief look at the history of exposure of a business function, then we'll take a more detailed look at the differences between the two most recent concepts, service-oriented architecture (SOA) and web APIs.

There is a clear evolution in integration architecture across the industry over the past few decades, with progressively greater degrees of exposure for a business function as shown in Figure 1.

Figure 1. Progressive exposure of a business function



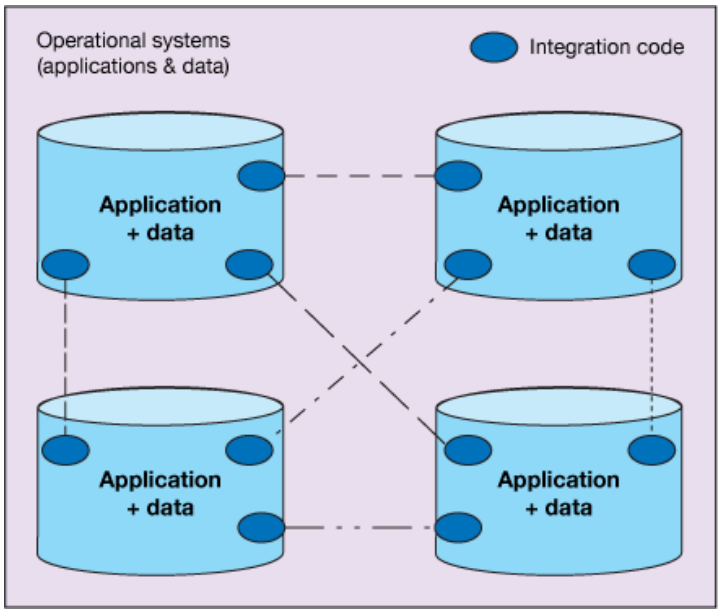
Our purpose is to understand the difference between SOA and modern business web APIs. In order to do that effectively, we need to have a clear picture on what SOA brings to the table.

Let's take a brief look at how things changed through the first three stages, up to and including SOA, then consider what web APIs add to that picture.

Point-to-point connectivity using “low level APIs”

Ever since enterprises have had applications, there has been a need to move and share data between them. Getting users to re-key information from one system to another ("swivel chair" integration) was unsustainable for most circumstances. This introduced the need for direct (point-to-point) low level connectivity between siloed applications. Often, it was impossible to get real-time responses, so data was typically sent asynchronously via files or one-way messages. A new serialization and parsing code was required on each side for every interface as shown in Figure 2.

Figure 2. Point-to-point integration



The different styles of lines between the applications depict that multiple different protocols were often required to achieve the different interfaces, further complicating the integration task.

Application programming interfaces (APIs), including transports, protocols, and data formats for real-time interaction, were introduced, where responses are gained directly back from the called system. This is, of course, the origin for the acronym of API that has gained new usage as "web APIs". We will clearly differentiate between the two in this tutorial by calling the original type as "low level APIs" and the new type as "web APIs". In day-to-day parlance, web APIs are now often referred to as simply "APIs".

Integration maturity is often measured using the Service Integration Maturity Model (SIMM). A point-to-point integration would be typical of SIMM Level 2. The SIMM is a maturity model, but let's be cautious with what we mean by "maturity". The techniques honed at each level within the model do not become obsolete when you move to the next level, they just get used more selectively. For example, even in a company implementing services at SIMM Level 4, there may still be a perfectly valid need for an occasional point-to-point or hub and spoke integration.

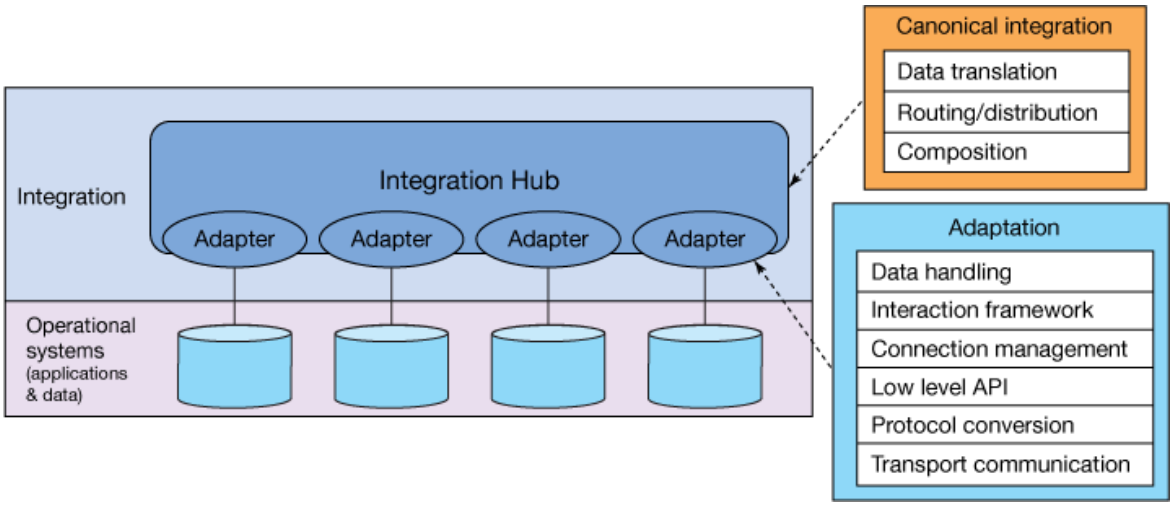
These low level APIs varied dramatically across platforms, which involve complex application-specific connectivity code to be written into in the applications on both sides of every integration.

Enterprise Application Integration

In the 1990's, integration tooling and runtime became more common place. They knew how to perform the connectivity and provided a central hub through which all integration could be performed.

This enabled a more "hub and spoke" architecture and significantly reduced the amount of proprietary integration code being written as shown in Figure 3. This is typical of SIMM Level 3, and was often termed *Enterprise Application Integration (EAI)*.

Figure 3. Hub and spoke architecture



This new tooling and techniques meant that you could reuse the connectivity within the scope of the integration hub - you only needed to work out how to connect to an application once. The same tools and runtime were always used for the job rather than the integration code in multiple languages and on multiple platforms.

Due to the radically different interaction styles between applications, they were typically not connected in real-time. More typically, an inbound adapter draws data from a system into a file or message based store, then an integration flow manipulates the data and passes it on to the target systems. This inevitably results in a significant amount of data replicated across systems, when the data is only really required for reference purposes. A real-time interface to the original system can reduce this replication.

Gradually, real-time interfaces to operational systems became more common, lessening the need for replication of data across systems. However, for a new system to use one of these real time interfaces, there was still some work needed to connect it to the hub. Indeed, many attempts at hub and spoke architecture only alleviated the point-to-point problem slightly, by bringing point-to-point coding into one runtime and one tooling. Unless the integrations were carefully designed for re-use, there was still significant new code needed to create a new interface.

We need a more standardized way to expose functionality from the hub so it can be re-used without extra work.

Service-oriented architecture

As standards for transport, protocol, and data format started to become more widely adopted in the early 2000's, such as SOAP/HTTP (often referred to more generically as "web services"), it became possible to expose services in a standardized way. This meant that for services, it was possible for requestors, who understood those modern standards, to make use of the services with minimal effort. Direct re-use of these exposed business functions now became possible. A suite of well-governed exposed services suggests a SIMM Level 4.

Any opportunity for re-use brings new benefits and also new challenges. Simply exposing a business function using SOAP/HTTP is not sufficient to ensure a robust service. Indeed, it leads to many challenges from unmanageable dependencies between systems to security exposures.

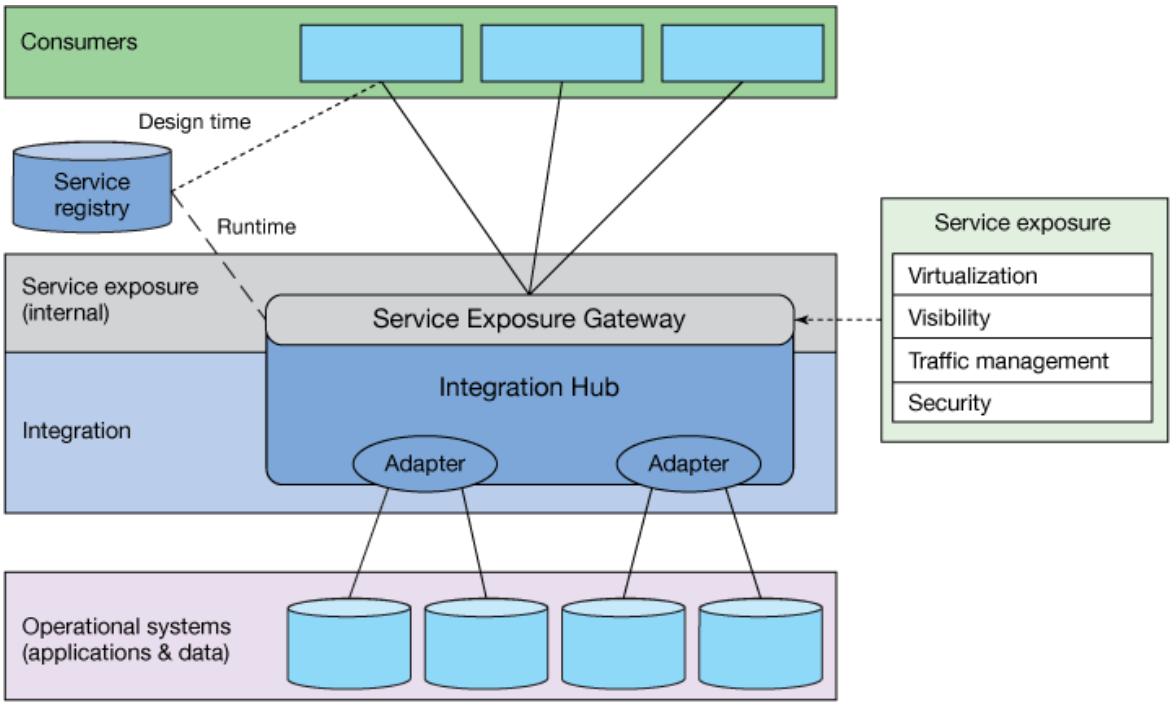
From a service exposure point of view, SOA is much more complex than just the standardization of protocols and data formats. To expose services effectively, you need standardization of the following aspects too:

- **Virtualization**: The consumer must call a *virtual* service endpoint that hides the complexities of how and where it is ultimately implemented. Conversion to the standardized protocol and transport is part of the virtualization, but services also need to provide standard configurable aspects, such as routing and data translation while continuing to provide the same *virtual* service to the consumer to minimize the effects of change.
- **Visibility**: If you are exposing core business functions, you need to manage and monitor them. To achieve effective monitoring at scale, it needs to be done in a standardized way across all services.
- **Security**: To make the services easy to consume and easier to manage, you need to standardize access control, identity management, and other key *security* aspects. Security is complex, and yet you need your services to be consumable. You need to reduce the variation of security models exposed to the consumer.
- **Traffic management**: How can you ensure that priority consumers always have access to the services they require with acceptable response time? What if you need to sacrifice one consumer temporarily in order to sustain another? How do you manage outages, whether planned or not? You need some form of configurable operation control over the service exposure point, enabling you to make adjustments without going through code cycles.

These are described in more detail at the beginning of this tutorial: Solution design in WebSphere Process Server and WebSphere ESB.

To do all of the above, you need to formally separate the service exposure capability in the architecture as shown by the Service Exposure Gateway in Figure 4. It may not turn out to be a separate runtime component in the final physical architecture, but it at least needs to be clearly delineated in the design. It must be possible to perform the virtualization, visibility, security, and traffic management requirements in a first-class way.

Figure 4. Service exposure



You will notice that we have deliberately *not* put the common SOA-related term, enterprise service bus (ESB), in the diagram shown in Figure 4. This is because opinions vary quite considerably as to what the exact boundaries are for an ESB. ESB is, after all, an architectural pattern, not a component description. Some say it is just the service exposure gateway, others include the integration hub. Some would include the adapters too, and there are many variants in between. There is plenty of literature describing the subtleties of the ESB pattern, but we have ultimately found it better to be simply clear about the specific individual components and their responsibilities.

Beyond the runtime components of SOA, there are also the governance aspects. If there are a large number of services, how do you decide and prioritize which functions to expose? How will people discover functions that have been exposed? How do you regulate the variations in the data models used? Some form of catalog of the candidate and current services must be kept to enable governance of the service lifecycle.

All of these concerns mount to a recognition that exposing services is not trivial. Yet, to simply expose capabilities as generally available web services leave you wide open to failures in manageability and security. In short, re-use comes at a price, and with that, comes the question of how to fund an SOA? No project with tight deadlines and budget constraints wants to be the one to build a service for the first time – at least not properly.

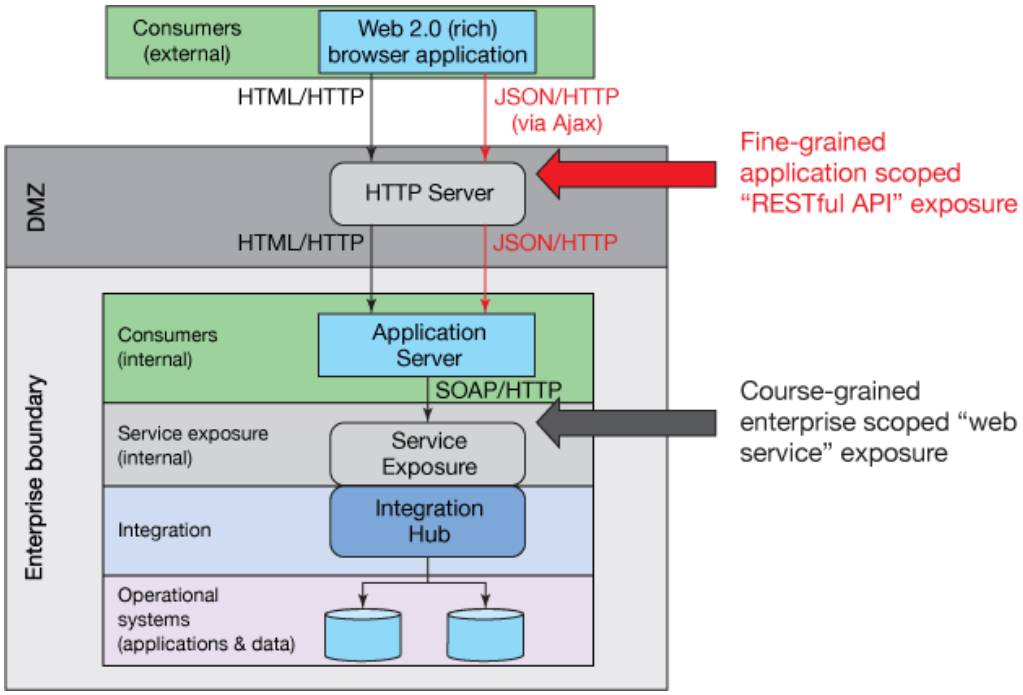
Couple with this with the fact that the standards required to enable the SOA concepts were being developed alongside the SOA initiatives, and were, therefore, immature. They were constantly changing while enterprises were attempting to implement them. It is easy to see why some SOAs struggled to gain traction. In many companies, SOA is confined to a specific domain of the business, or there are only a small number of core services actually in play.

Introduction of JSON/HTTP interfaces

Browser-based applications became more sophisticated, and mechanisms were introduced to write richer, more responsive web pages. These took advantage of the maturing client-side scripting capabilities of browsers, and also their ability to perform background HTTP requests using techniques, such as AJAX, to retrieve data without interrupting the user experience with a page reload.

The web pages were typically requesting data specific to the page via the page's associated web server. SOAP/HTTP requests typical of SOA were hard to handle in JavaScript, and too heavyweight to send over an often low bandwidth Internet connection. It quickly became popular to make more granular data requests and, if possible, change to the JSON data format native to JavaScript as shown in red in Figure 5.

Figure 5. Fine-grained exposure for rich browser applications



Free from the restrictions of the SOAP standards, these interfaces can consider alternative ways to simplify how the "actions" or "operations" to be performed on data were to be represented. In an interesting throwback to some of the earlier roots of the web, the original intent behind HTTP protocol was unearthed. Significant aspects of the HTTP standard were designed around the architectural principles of Representational State Transfer (REST) documented by Roy Fielding in 2000. From this, a more simplistic entity based style of interaction was derived. The interaction style recommends the use of the common HTTP verbs (POST, GET, PUT, DELETE) in a similar way to the common database interaction verbs (create, read, update, delete). The worldwide web recognizes these verbs in a first class way in order to provide implicit benefits such as caching. It also uses the URL path to navigate the relationships between the data entities.

In a rather over simplified example, you could imagine the addition of an item to an order might be performed by an HTTP "POST" to a URL that looked something like the following URL:

```
https://www.mycompany.com/orders/123456/item
```

The JSON formatted data in the body of the HTTP request looked as follows:

```
{ "title" : "Romeo and Juliet",
   "note" " "Special Edition",
   "quantity : 1,
   "price" : 9.99 }
```

Where the URL describes the specific data entity (typically referred to as the "resource"), and the use of the HTTP verb "POST", meaning it is a "create" operation for a new order item. To carry the same information in SOAP, it might have looked more like the following:

```
http://www.example.org/ordermanagement HTTP/1.1

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://..." soap:encodingStyle="http://...">
  ...SOAP headers...
  <soap:Body xmlns:m="http://www.example.org/ordermanagement">
    <m:AddOrderItem>
      <m:order orderid="123456"
        <m:item>
          <m:title>Romeo and Juliet</m:title>
          <m:note>Special Edition</m:note>
          <m:quantity>1</m:quantity>
          <m:price>9.99</m:price>
        </m:item>
      </m:order>
    </m:AddOrderItem>
  </soap:Body>
</soap:Envelope>
```

These JSON/HTTP based interfaces provide some useful simplifications in comparison to the more increasingly complex SOAP standards that came before them. However, SOAP has a broader set of standards, which can accomplish many things that these interface cannot. They are used by a different audience and not all of those standards are necessary in that space.

There were, at least initially, some limits to the reusability of these new interfaces. Due to the same origin policy implemented by browsers, web page based applications found it difficult (though not impossible) to make HTTP requests to interfaces offered by other companies. This means that the most common initial use of these JSON/HTTP based interfaces was between a company's web pages and their own enterprise data. However, techniques such as proxies, JSONP, and standards, such as CORS, ease these restrictions, and enabled these interfaces to be re-used more widely and for "web API" to become a reality.

What is a web API?

There is no formal definition for exactly what is meant by "web API", just as there wasn't for "web services" before it, but let's do our best to nail it down.

Simplistically and loosely speaking, a "web API" typically refers to functions and data exposed in the following way:

- Exposed over HTTP(S)
- Uses the HTTP protocol "RESTfully"
- Uses JSON as the data format
- Is available over the Internet

For anyone creating a new web API today, this is likely where you start. However, this definition is over simplistic on a number of levels:

- **Not all web APIs use JSON:** Most APIs use JSON as the data format, but some provide XML as an alternative format, or even exclusively. In theory, anything that HTTP can respond with could be valid. If you include MIME types, this could mean PDF files, for example, though this broader usage is less common.
- **Not all web APIs are public:** As we will see in a later section, APIs are not only exposed and used on the public Internet. However, it is fair to say that the Internet usage has driven much of the agreement on style, usage, and the supporting products and frameworks.
- **Not all web APIs use HTTPs' RESTful properties directly**: There are many Internet facing SOAP/HTTP interfaces and it would be hard to deny that these are also web APIs in some form. They are probably less "RESTful" and more heavyweight to use. However, many SOAP/HTTP interfaces have subsequently introduced JSON/HTTP "RESTful" counterparts.
- **Few web APIs are completely RESTful:** The use of JSON/HTTP in web APIs means that they are certainly more RESTful than what came before them. They are, therefore, often referred to as "REST" interfaces. However, in reality most only adhere to a subset of the REST recommendations described in the original material on the subject. A common complaint leveled at APIs claiming to be RESTful is that they rarely provide links recommended by the HATEOS approach.
- **HTTPS is strongly recommended:** HTTPS is certainly preferable, and many would say mandatory for web APIs. Payloads often contain private data, and credentials used to access the web API are usually confidential.

So, a new, more lightweight protocol and interaction style is available, but this alone does not warrant a revolution in real-time data integration.

What was the trigger for web APIs to come of age?

The significant shift came around 2007 when smartphones with easy to access "app" stores became mainstream. Mobile application ("app") development became commonplace and accessible to a huge audience of developers. With some notable exceptions, apps can rarely do much on their own. They need to interact with the world of data around them. Developers can write much more powerful apps if they had simple ways to incorporate access the data and functionality provided by other companies.

It was not just demand from mobile app developers, but developers of richer web sites also needed broader and easier access to data. However, mobile typically brought a high volume of

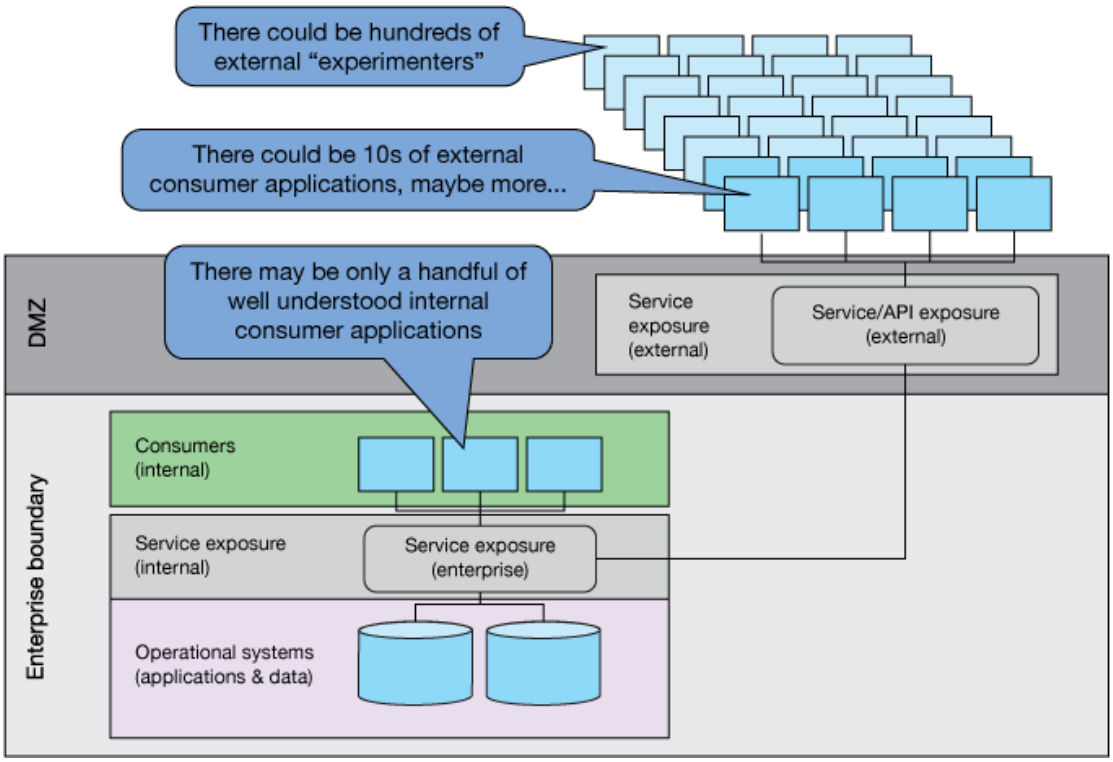
new web API consumers. They were not held back by some of the security restrictions that had challenged the use of APIs in browser-based applications. So, you can say that the introduction of web APIs came about from to two key influences: demand and capability.

- **New ways of making money:** A new profitable funding model driven by, but not limited to, the prevalence of a new generation of mobile service consumers in the form of applications for phones, tablets, watches, and more, all needing access to live data and function.
- **Matured capability:** The standards, technology, and methods for exposing business function have matured by a decade of work on re-usable service exposure. For example, exposure protocols and data formats have been trialed and matured. Exposure gateways for APIs are now available as a first-class component with a well-understood remit.

How are web APIs different from what came before?

It is in the new demand, and its associated funding model, where the big difference lies. The audience for the exposed business functions are beyond the walls of the enterprise. As shown in Figure 6, SOA services were more typically exposed *within* the enterprise, generally based on a funnel of projects and their respective known requirements. Web APIs are typically exposed *externally*, to an often unknown and potentially enormous user base for often highly innovative and unexpected uses.

Figure 6. New factors affecting exposure of services outside the enterprise



If a web API can provide data that is useful to one application, it is probably useful to others. Put those services (or APIs) on the web, and there is suddenly a whole Internet of potential consumers of the web API who can reach out to many previously untapped customer segments. There lies the

new funding model. There is an opportunity to monetize these exposed business functions. The web API becomes a new "product" provided by the organization.

The return on investment might be in many different forms:

- **Direct revenue:** For example, a web API for purchasing goods.
- **Indirect revenue:** For example, earning a commission by providing aggregation services to other providers.
- **Cost saving:** For example, apps that enable self-service in order to scale down an expensive customer service.
- **Marketing:** For example, putting product information in front of a wider customer base.

What is for certain is that new markets can be reached by crowd-sourcing the innovations of application designers beyond the walls of your enterprise.

How will you cater for this significant change in the requirements for your integration architecture?

How is the exposure componentry different for web APIs?

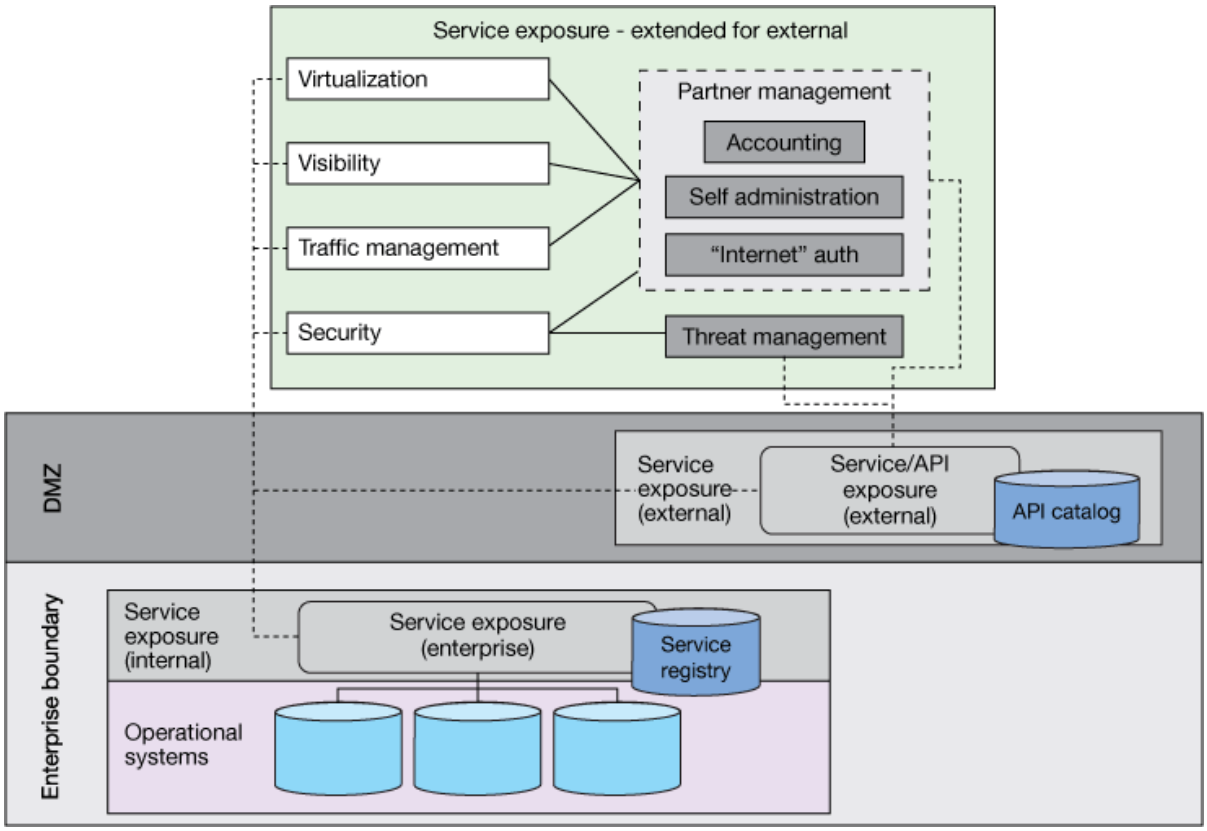
Based on our earlier definition of web APIs, you can see there are three key aspects that will fundamentally change when you expose external web APIs compared to enterprise services:

1. **Beyond the enterprise:** Web API consumers are not part of the enterprise, so they are beyond your direct influence and control.
2. **Numerous consumers:** A significantly higher number of potential consumers of your APIs than you have for an enterprise service.
3. **Competitive marketplace**: Consumers have alternatives provided by other companies if your web API does not meet their expectations. Within the enterprise with SOA, they may have only had one option.

These key differences result in a huge number of amendments in how you architect and design web APIs. In this tutorial, we will focus primarily on the differences in the architecture. Architecturally, you clearly still need some form of exposure gateway, but the requirements for that gateway have some new elements.

As you will recall from earlier in the tutorial, re-use always comes at a cost. As shown in Figure 7, your exposure capability needs to be extended beyond the core SOA requirements.

Figure 7. New capabilities for exposure APIs outside the enterprise



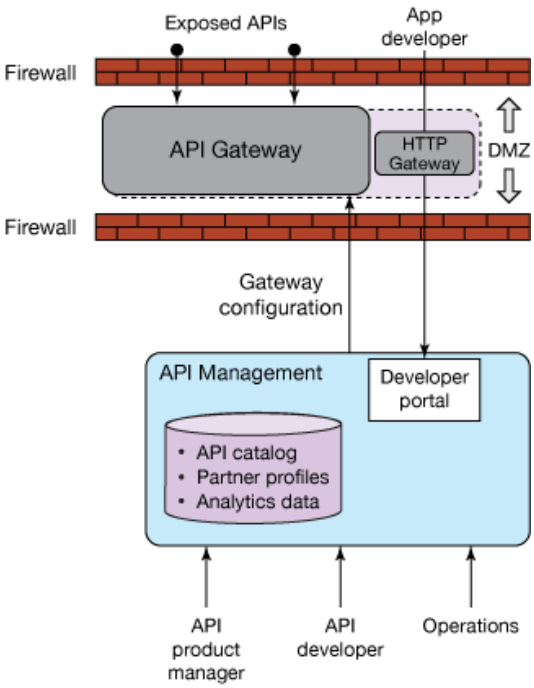
Let's have a look at two of the primary aspects of those new challenges:

- **Partner management:** You now have a potentially enormous pool of mobile application designers that may want to experiment with your web APIs. If you do not make that extremely easy and attractive, your web API "product" will soon be overtaken by your competitors. How do you set up new relationships with these new partners? How do you keep a track on who has access to the functions? You may have external parties with a dependency on your web APIs as a fundamental part of their business. How do you establish and monitor what level of service they need? Partner management needs to be a first-class function provided by the web API exposure components. Due to the volume of potential partners, it needs to be self-service. It also needs to recognize partners and to monitor and control their usage according to their agreed entitlement plans.
- **Security:** Clearly, exposing web APIs over a public medium such as the Internet means a whole new level of security concerns, from the various payload borne attacks, such as XML threats to denial of service attacks on throughput or connections. You must also reliably authenticate your partner's applications to control their service levels effectively.

This increased complexity has resulted in the emergence of what is now known as *API Management*. Web API Management is an architectural intent rather than an individual component, although there are, of course, products specifically designed for that intent. It enables an organization to simply, yet securely, expose and manage a web API. It combines a more robust and secure gateway with new capabilities relating to partner management.

Figure 8 shows the primary components required to implement API Management, and the typical roles involved.

Figure 8. Example of a component model for API Management



All of these roles were loosely present in SOA initiatives in one form or another, but they were often less formally implemented. The public facing nature of web APIs has forced their maturity. The typical set of roles are:

- **API Product Manager:** This role establishes marketable web APIs, prepares and administers "plans" for their use, and evaluates the success of the web APIs using historical analytics.
- **API Developer:** This role provides the substance behind the web API façade by configuring the connectivity and data mapping through to the backend systems that provide the actual data and function.
- **Application Developer:** This role explores the web APIs on offers via an "API Portal", and signs up to use them via one of the plans defined by the API Product Manager.
- **Operations:** This role monitors and manages the web APIs on a day-to-day basis, ensuring they are meeting the service levels defined by the plan.

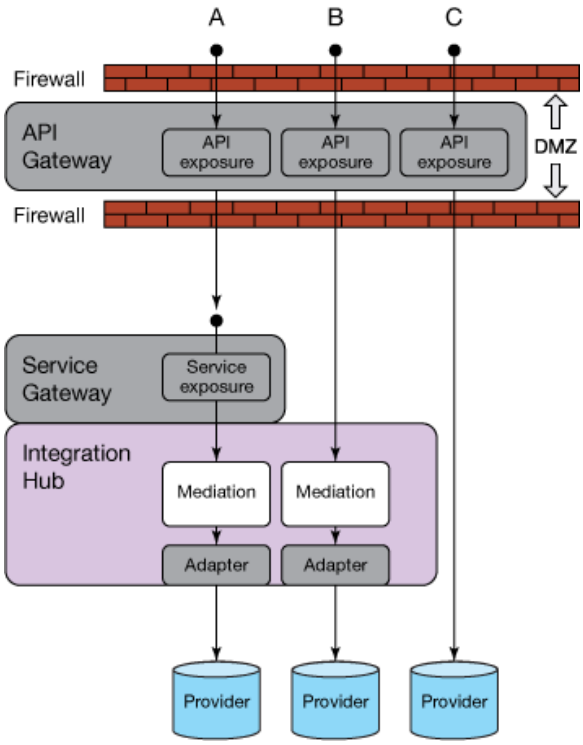
What is the architectural “substance” behind the API façade?

The web API Gateway is just the façade or exposure point of the web API. It provides none of the actual function or data provided by the web API. This brings us full circle to the integration architecture that has evolved within the enterprise - the growth from siloed applications through point-to-point communications and hub and spoke middleware, to potentially implementing an SOA.

Indeed, there is a strong dependency on how an enterprise has evolved when it comes to deciding where the underlying implementation of a web APIs should come from. Figure 9 shows examples of the most common options:

A. Re-exposing an existing enterprise service
B. Exposing a new integration mediated through the hub
C. Exposing an interface already offered by a provider system

Figure 9. Examples of options for web API implementation



It is tempting to think that re-exposing existing services (Option A in Figure 9) is the most common. After all, SOA has been around for more than a decade, surely most companies will already have a suite of services available. Exposing those will be the most efficient way of monetizing that prior investment in integration. While there are certainly some organizations that have done this, it is less common than might be expected due to any number of the following reasons:

- **Integration maturity:** As noted earlier, it is common for services to have only been exposed in specific portions of the business where they were of value. For the other parts of the business, other integration patterns, such as hub and spoke or even point-to-point, may have remained perfectly adequate.
- **Granularity:** Since enterprise services were typically created based on known business requirements, they are often fairly coarse-grained, bringing back a relatively large data graph, and perhaps including many child objects. These operations are too heavyweight for the highly responsive applications required on mobile devices, especially taking into count the often variable bandwidth of the device.

- **Security:** Where enterprise services perform an updating operation, it is often performed in an enterprise specific way; for example, making assumptions about the trustworthiness of the caller, the safety of the channel, or using internal only mechanisms for security tokens.
- **Relevance:** Data and function, that are interesting or marketable inside the company, are often either irrelevant or inappropriate for external exposure. What we are often seeking with web APIs is a completely new market opportunity; a new concept, probably exposing completely different data.

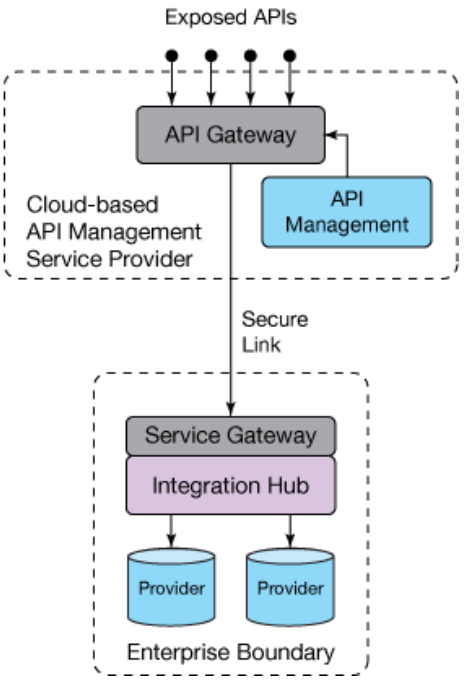
As a result, Option B in Figure 9 is likely to be common and web APIs are implemented using integrations. Perhaps they are re-using components and adapters within the integration hub itself, but not re-using existing services.

Option C is currently rare because the desired web API is almost certainly different than the existing application. Web API gateways typically have a limited data translation capabilities However, once the integration logic goes much beyond basic data mapping and simple aggregation, the web API Gateway is architecturally an inappropriate place for this logic.

Cloud-based API Management

In the current era, where there is a desire to reduce the internal infrastructure footprint, enterprises are looking for capabilities that can be more easily sourced from cloud-based providers. API Management is a good example of this. Both the web API gateway and API Management capability have clear responsibilities so they can be easily separated from the internal architecture as shown in Figure 10. Since the aim is to expose them to external parties, hosting the web APIs from a cloud-based provider makes some sense.

Figure 10. Cloud-based API Management service provider



Although relevant to any enterprise, this cloud-based API Management particularly suites companies that are "born on the web". For example, a start-up company that has chosen to have no internal infrastructure and host everything in cloud-based environments. Cloud-based API Management enables them to provide a single unified exposure point for consumers to find their web APIs, regardless of where they are hosted in the cloud.

With this approach, you need to consider the following:

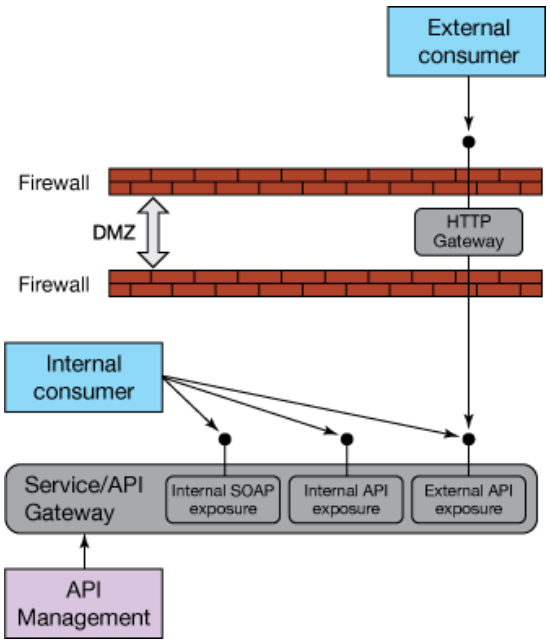
- **Access control:** How do you safely provide the cloud-based API Management service access to your internal middleware or indeed operational systems? Clearly, some form of secure connector is required, but how much control are you prepared to put in the hands of an external party?
- **Latency:** Consumers of your web APIs will now have to perform two hops over the Internet before they reach your site – one to the web API provider, and then another to your organization. Web APIs tend to be more "chatty", meaning you often do more calls to achieve the same purpose. As such, a greater portion of end users wait time relates to the latency of the communication. Depending on the global reach of the service and how busy the web API interactions typically are, this extra latency could be significant. API Management solutions often provide caching options to reduce this latency for requests that are appropriate for that pattern.

APIs within the enterprise – the next phase in SOA?

The new style of "RESTful" JSON/HTTP interfaces are more lightweight, well-suited, and supported by modern programming languages and frameworks. The API Management gateway products are increasingly well-matured. Why use all that goodness within the enterprise too? Many now see an internal API Management layer as an alternative, and in some cases, a more effective way of exposing data and function within the enterprise – an extension to the Service Exposure Gateway shown back in Figure 4.

Figure 11 shows a minimalist architectural configuration that enables a single point of administration of both internal and external APIs. This has the added advantage that internal consumers can directly consume the externally exposed APIs without having to route out to the Internet. In reality, enterprises will often still prefer to have separate internal and external API gateways for a more robust separation of concerns.

Figure 11. APIs for internal and external consumers



So for some, APIs have become the next stage in their journey towards a service-oriented architecture within the company as well as the opportunity they present for external exposure.

A new form of B2B

A further case worth considering is where the maturing of web API technology and practices are used to provide business to business (B2B) communication. B2B interfaces are not new by any means. For decades, businesses have been exchanging data using a variety of standards, such as those relating to Electronic Data Interchange (EDI). Due to their specialism for specific industry needs, the standards and the tools used to implement them are necessarily deep and complex. Many enterprises need a more lightweight alternative for interchanging data, but they still need some core requirements, such as the security capabilities and self-administration for partners. Even if a company does not plan to provide APIs to the general public, leveraging the security and partner management aspects of API Management can be highly valuable to enable straightforward private interfaces with specific business partners.

Beyond web APIs

It would be difficult, and probably foolish, to try to guess what comes next for integration architecture with the pace of technology in this space in recent years. It seems likely, however, that it will involve a further nod to the near term realities of the mobile user. An important factor is the desire to be "always online" that still translates to "intermittently connected" for the broader community.

This implies a resurgence in message-based event-driven interactions. We are already seeing more event subscription-based interaction models. For example, all the major mobile vendors have mechanisms for pushing events back to devices. Sensors are now capable of emitting events in

more widely recognized protocols, such as less proprietary coupling between the sensors and the subscribers to their events. The "Internet of Things" is bringing sensors in everything from wearable technology, to connected cars, and to a different genre of applications. These may be less focused on retrieving specific data from operational systems, and more interested in tuning into topics of interest and intelligently interpreting the events they receive.

Conclusion

Looking back to Figure 1, you can see how integration architecture has seen a progressive exposure of business function to more public audiences, and a maturation of those capabilities used to perform that exposure. API Management is the latest in that line, but it is important to recognize that the patterns and techniques and concepts developed along the way, such as hub and spoke architecture and SOA, are still relevant and appropriate in the right circumstances.

Acknowledgments

This tutorial is a result of regular and ongoing collaboration on integration topics with **Brian Petrini**.

Simon Dickerson, **Andy Garratt**, and **Matt Roberts** also provided extremely valuable input to the tutorial.

Related topics

- IBM API Management Family Page
- Service integration maturity model (SIMM)
- SOA Reference Architecture
- Roy Fielding's dissertation on REST
- The Enterprise Service Bus, re-examined
- Solution design in WebSphere Process Server and WebSphere ESB: Part 2