eBook

APIs are Different than Integration

Internet & Enterprise Technologies Unite in a New Enterprise Foundation

By Ed Anuff

apigee

apigee

Table of Contents

apigee

Executive Summary

Many developers and architects see APIs as an evolution and continuation of the integration-based architectures that have long been in use within enterprise IT.

However, this is a limited view that misses the larger realization that APIs have become a foundational technology for the development of robust and scalable enterprise applications. Apps are built using APIs on the front end to connect the user experience (UX) tier (whether HTML5 or native mobile clients).

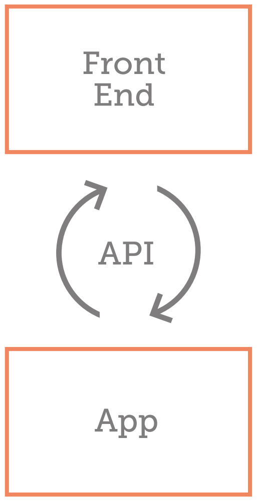
Apps use APIs on the backend to connect to data and services. Apps use APIs on the sides to enable other apps to connect to their internal data and processes. And finally, apps themselves are built out of interconnected “micro-services”, which are wired together via APIs.

In this eBook, we explore these ideas in detail as well as examine some of the important implications of the movement to the API-centric architecture that is underway within enterprise application development today.

All Development is API Development

All development is API development because all software is built as services. Rather than using web frameworks that invoke services and produce web pages, today's applications are built by consuming and producing APIs.

The majority of enterprise software development efforts are focused on building web applications using web frameworks. Java Enterprise Edition, Microsoft .NET, Ruby-on-Rails, PHP, and a variety of other technologies are used to implement web applications using model-view-controller (MVC) design patterns and page template technologies. For a decade, starting in the late 1990s, an entire generation of developers has spent the majority of its time building applications in this model.



All development is API development
All development is API development because all software is built as services. Rather than using web frameworks that invoke services and produce web pages, today's applications are built by consuming and producing APIs. Mobile and HTML5/JavaScript have accelerated this.

APIs move to the front end

The advent of Web 2.0 brought a renaissance to rich client development. The incompatibilities and inconsistent implementations of JavaScript started to ease as more and more users updated to the latest browsers.
At the same time, renewed competition in the browser market drove more innovation and acceleration of the implementation of web standards. HTML5 and JavaScript made it possible for much of the interaction to run

in the browser rather than to be orchestrated via page-based flows. It became necessary for the web application developer to create APIs to allow communication between the browser and the server. The early uses of this were for simple interactions, such as the ubiquitous "type-ahead" auto-completion, where the user is prompted with words, phrases, or names as they enter data. Each keystroke on the browser invokes an API call to the server to retrieve the list of words with which to prompt the user.

Over time, these techniques accumulated, with more and more functionality that had previously been implemented in page templates becoming replaced with API calls. The end-result is what's now called the "single page app" or SPA. At this point, no user interactions are serviced via page templates. A single web page is served and all interactions are handled via JavaScript, which generates a user interface via HTML5 techniques and libraries such as jQuery. Nearly every user interaction generates API calls to support these interactions.

As this model of web development has become common, traditional page-based web analytics has been hampered, and developers have had to log interactions to APIs if they've wanted to have any visibility into the use of their applications. While this was a burden at first, and introduced a crisis to the business functions that depended on the usage data to drive their decision-making, developers soon found that moving to API-based interaction analytics gave them greater abilities to capture and analyze web usage. It became possible to use JavaScript to observe, in much greater detail than before, what users were doing within the applications and where they were spending time. These analytics are then transmitted to analytics APIs in a constant stream of interaction data.

Mobile forces the issue

With the advent of mobile, the API-driven UX becomes non-optional. While the web application developer could choose the interactions to support via HTML and JavaScript versus those driven via traditional page templates, there was no such choice possible for mobile developers. The mobile app represents a return to the more strongly delineated client-server model of development, where all interactions happen client-side and a constant networked communications mechanism between client and server must be maintained.
In the age of mobile, that communications mechanism is APIs. In some cases, the same developer is charged with building the server-side and client-side interactions,

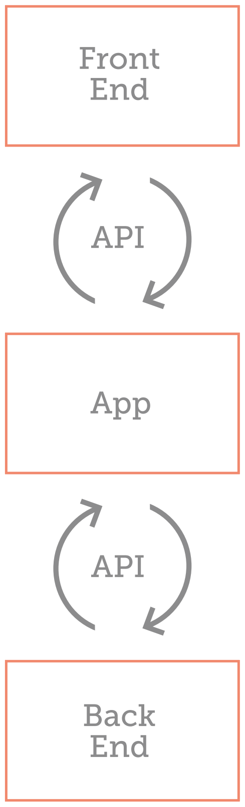
apigee

but in other cases, such as for companies that needed to support native iOS and Android applications, it became necessary to have specialists building each client implementation. Further, these specialists are often external to the company: they can be contractors, digital agencies, or system integrators. This means that the application development effort needs to have an API project at the heart of it, which in turn means that application developers not only need to know how to consume an API, but have to understand how to produce one as well, with all the attendant issues such as API design, API security, and API scalability.

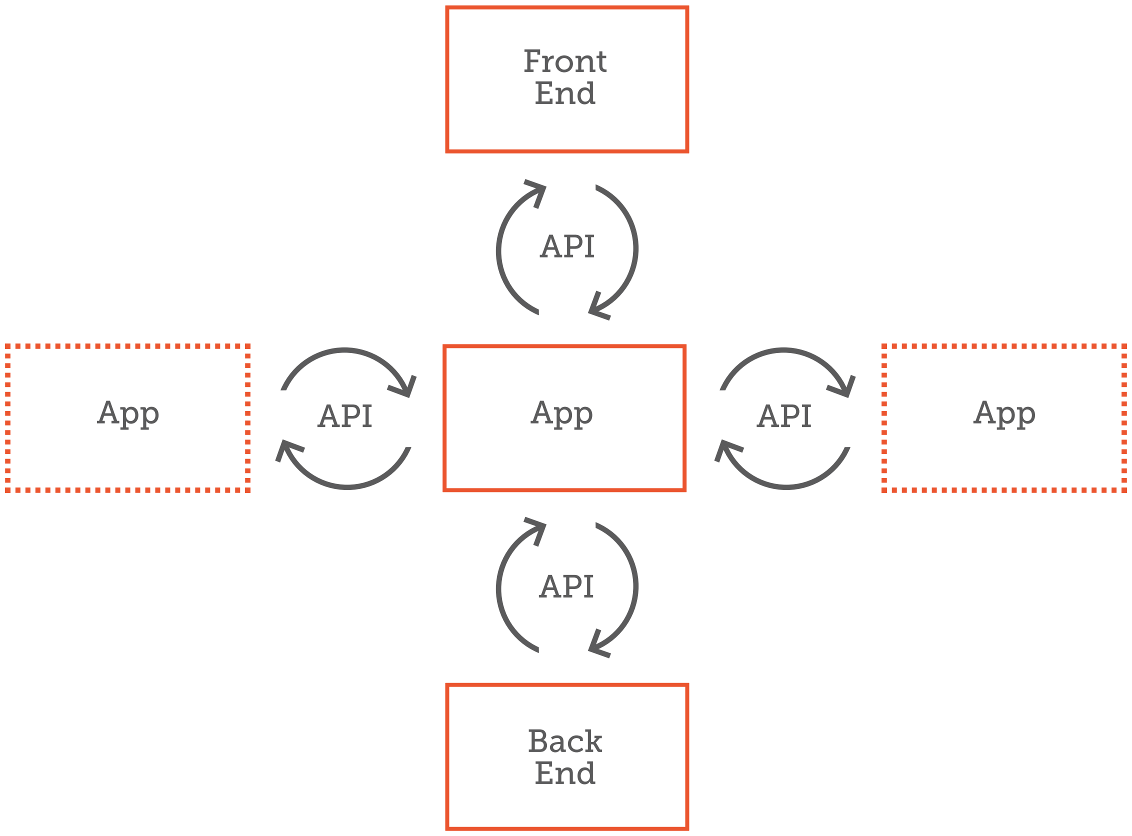
APIs open enterprise applications

It's not a great leap to open the rest of the application functionality via APIs, making it possible for applications to leverage and incorporate functionality in other applications as needed. Integration architectures assume that applications are developed without considering whether to enable other applications access to their internal data and processes. Before the conventions of APIs were widely adopted, this was a prudent strategy. Trying to support application-to-application integration was an undue burden for the application developer and was usually left as a project for another team to take on.

APIs Open Enterprise Applications
The widespread adoption of REST, JSON, key-based access control, and the other basic conventions of API design means that developers can easily build APIs into their applications and support their usage without undue distraction from their central mission of application delivery.



The widespread adoption of the basic conventions of API design—RESTful design, JSON-based data, simple versioning, and key-based access control—means that developers can easily build APIs into their applications and support their usage without undue distraction from their central mission of application delivery. And because application delivery already requires an API-centric architecture to enable rich HTML5/JavaScript and mobile clients, APIs will be essential elements of the project scope. The result: when the time comes for application-to-application integration, the need for separate integration middleware is obviated.



apigee

SOA Gives Way to Micro Services Everywhere

The dream of SOA has become reality in micro services architecture, with applications decomposing into sets of fine-grained services. The motivations haven't necessarily been about generic service reuse as much as about best practices for building scalable and reliable applications using IaaS and PaaS.

Applications have embraced APIs on the front end for connecting to rich clients; on the backend for integrating with internal systems; and on the sides for enabling other applications to access their internal data and processes. The next phase of application architecture is to explode the application into a set of component services, linked together via APIs. This is the concept of the micro service architecture, or MSA.

Resilient and scalable cloud deployment for componentized applications

Modern Java applications are heavily componentized. Using frameworks like Spring or Java CDI, applications are assembled from a set of service components wired together at runtime. The advantages are that large complicated applications can be broken down into components that can be developed by disparate teams, applications can leverage and reuse existing components, and components are interconnected without fragile complex dependencies or tightly-coupled linkages.

However, while the benefits of the software development principles behind these practices are inarguable, the implementation of this type of architecture has generally occurred in ways that are antithetical to the deployment of resilient and scalable cloud deployments. The problem has been that enterprises have embraced application server deployments that have generally sought to run as much of a componentized application as possible within increasingly large server instances.

Heavyweight application server architectures, as exemplified by the typical WebSphere or WebLogic deployment, are exceedingly expensive to scale elastically. The idea of firing up new server instances on-demand based on fine-grained assessments of load and traffic was not a central consideration in the design of such servers. The loading up of all components in a single JVM process within an application server means that the entire application (and its container app server) is either "up" or "down"—the latter condition being a catastrophic application failure.

Further, not all components in the application are under equal load, yet they all must be scaled together. It's not easy to allocate additional computing resources to a single component, nor is it easy to protect the other components from one that is consuming large amounts of CPU or memory resources. Of course, there is an assortment of mechanisms for remote invocation of components that obviates the need to run them all locally. However these mechanisms typically attempt to hide the fact that the invocation is remote from the application developer.

The intent is to make it easy to invoke code that runs locally or remotely, but the result is at best "magic," and at worse a system that is impossible for a developer to debug. As a result, most developers distrust such mechanisms and simply opt for provisioning very large server instances where they can run as much as possible inside the safety of the JVM.

However, because of the limitations of this approach, most highly scalable and available cloud applications tend to move to a networked component model where applications are decomposed into "micro" services, which can be deployed and scaled independently.

Decomposed services and polyglot enterprise development

Rather than leveraging the more complicated traditional enterprise mechanisms (whether the legacy RPC approaches of CORBA and RMI or the cumbersome web services protocols such as SOAP) many developers are finding that the same lightweight API services that have proven to be resilient, scalable, and agile for front-end, back-end, and application-to-application scenarios can also be leveraged for application assembly. This is the essence of the micro services architecture.

The Essence of Micro Services Architecture

Today's developers eschew complicated traditional enterprise mechanisms, and find instead that the same lightweight API services that have proven to be resilient, scalable, and agile for front-end, back-end, and application-to-application scenarios can also be leveraged for application assembly.

One final point about the emergence of the API-based MSA architecture. In recent years, enterprise development has moved to become "polyglot", as PHP, Ruby-on-Rails, Python, and node.js all find a home in the enterprise toolbox. Because of this, many Java-only or JVM-only architectural practices have become less compelling for many organizations. This is one of the reasons why the simple REST/JSON API, which can be built or consumed from all popular languages and frameworks, has become the foundation for modern architectures. It is also the reason why the new "API generation" of developers doesn't see a connection between these practices and classic SOA.

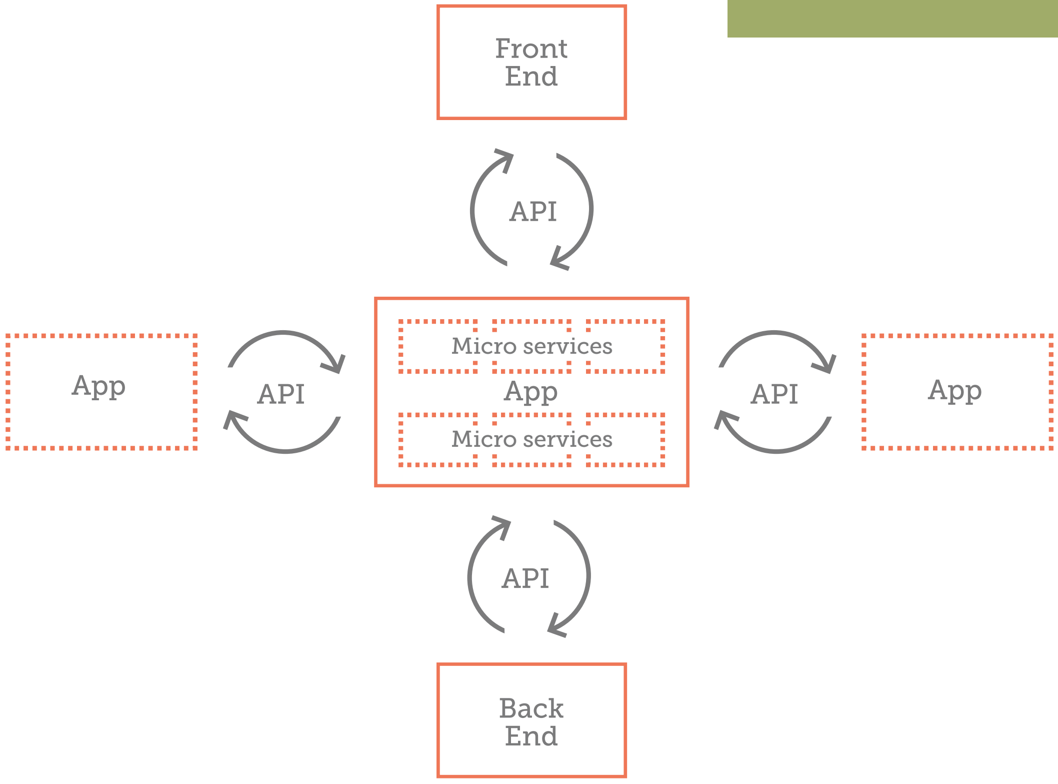
Facets of Modern API and App Architecture

App-to-client: Apps built using APIs on the front end connect the UX tier.

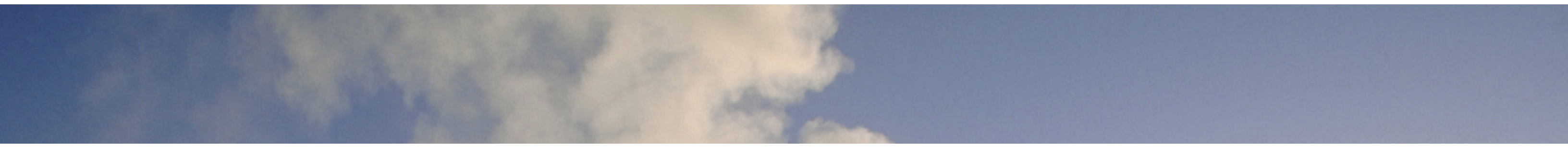
App-to-backend: Apps use APIs on the backend to connect to data and services.

App-to-app: Apps use APIs on the sides to enable other apps to connect to their internal data and processes.

Exploded apps: Apps themselves are built out of interconnected "micro-services" wired together via APIs.



Services Governance Does Not Scale



Services creation is a natural part of the software development process; it happens ad-hoc and it's prolific. API design and development is every developer's job. A centralized services governance process owned by a special architectural team in IT cannot maintain an iron grip on agile and decentralized API-first architectures.

With the proliferation of API services as the connective tissue between applications, frontends, backends, and even within the application architecture itself, one might think that service governance would have become paramount in importance. Actually, the reverse has happened. The majority of APIs are developed outside of most service governance processes.

Culture and history can explain—at least in part. Much of the early SOA implementations were done by central architecture teams who were enamored of the elegance of a unified architecture but who were removed from the realities of day-to-day application development.

In addition, many of the concepts of distributed computing that architecture teams worked with were still relatively unfamiliar to most application teams. However, we're now at a point where an entire generation of developers has come of age in the era of the internet.

The emergence of virtualization (and its descendants, IaaS and PaaS) has led to the final breakage of the centralized service governance model. Easy access to server resources has allowed application developers the freedom to build API services as they see fit. The micro service architecture described in the previous section is the direct result of the proliferation of virtualized server instances. This has meant that putting an API into production can often be done with little or no oversight.

It would be easy to criticize this API proliferation as leading to an unmanageable situation. There are legitimate concerns with such practices. Many APIs built in this way will not be secured consistently across the organization. Not all of the APIs will be designed as easily reusable services and more than a few will be used for tightly-coupled communications.

Services Governance Does Not Scale

API governance has emerged as a new area of focus in these situations, separate and distinct from SOA governance. API governance concerns itself with providing standardized conventions for documentation and consistent security and access control mechanisms. It exists in support of the application teams rather than the centralized IT resources and as a consequence is not prescriptive except in a few vital areas, such as defining standards for security mechanisms including OAuth.

API Governance for Application Teams
API governance concerns itself with providing standardized conventions for documentation and consistent security and access control mechanisms. It exists in support of the application teams

Integration "Patterns" Are Not Required in the DevOps World

New API development happens in a DevOps model, leveraging IaaS and PaaS, on either public or private clouds. There is no need for a separate "integration" product owned by a special ops group in IT in order to satisfy the new API use cases. Integration models rooted in appliance-heritage products have no place in the automation-centric DevOps processes.

Integration technologies are foundational components of many IT architectures. Orchestration and transformation of complex back-end legacy systems are necessary functions in many settings. In addition, there is ample evidence that SaaS services, particularly first-generation services such as Salesforce.com, cannot avoid the necessity of complex integration processes in order to connect to existing enterprise systems. However, the new API use cases (especially those driven by mobile), as well as new API-centric or micro-service based development efforts, typically have little need for integration server technologies in order to be implemented.

Speed and agility force the issue

Although integration vendors have long touted the ease of configuration of their products as key benefits of using their offerings, the reality of their usage is that they are often deployed in "set and forget" scenarios. Integrations are built and then left alone until some form of breakage requires them to be updated. As a consequence, most integration servers are attended to on an infrequent basis and they are used by the few administrators within IT who are knowledgeable about their capabilities. For most enterprise developers, the integration servers are a black box, and while application developers may be clients of the services they expose, they seldom, if ever, get involved in the configuration of the integrations exposed.

Many vendors have delivered their solutions in the form of appliances—as either virtualized appliances or physical boxes running in the datacenter. This delivery vehicle together with the fact that that many of the users of these technologies preferred them in appliance form demonstrates that their role is not within the mainstream development processes.

The limitations of these approaches are manifest. Not only have these products been impossible (in the case of a physical appliance) or at least difficult to deploy in public or private cloud environments, but their usage is proving to be of questionable value, especially within the increasingly common practice of DevOps.

Ops at the Speed of Dev
Integration models rooted in appliance-heritage products have no place in today's automation-centric DevOps processes.

DevOps (a portmanteau of development and operations) is a software development method that leverages automation to enable agile development and streamlined deployment processes to accelerate the release of software to production. Initially very popular within internet companies that needed to break down the traditional walls between development and IT operations in order to deliver cloud-based applications at scale, DevOps has also become a fixture within enterprise IT for many of the same reasons. Enterprises are embracing the same cloud techniques as internet companies and DevOps has proven an efficient way to reduce costs by sharing responsibilities across development and operations teams.

APIs: Ops at the speed of Dev

The DevOps model only works when, as much as possible, the resources within the application tier, including APIs and the systems they connect, can be managed by the same common set of DevOps automation tools (such as Chef, Puppet, Ansible, or Salt, as well as version control services such as Git). To make this automation possible, developers are constantly looking for ways that application configuration can drive all aspects of the deployment process, and integration servers, which frequently bring manually-configured dependencies to the application delivery process, are unwelcome sources of breakage. Developers strongly prefer coding to a set of APIs and taking responsibility for adapting their applications to those APIs rather than introducing black boxes into the dependency chain.

Consider that the principal use cases where API development occurs today occur in the application teams who are building the front-end technologies most likely to be aligned with DevOps. Because most web technologies are updated and need to be deployed at a faster cadence than would be possible in traditional developer-to-operations handoffs, deploying web technologies was one of the first places where DevOps processes became necessary.

Since mobile is inherently API-centric, this further reinforces the idea that these use cases should not be addressed by heavyweight integration products.

apigee

APIs Make Agile Data Possible

The final key difference between the API-centric architecture and one that depends on integration technologies is in the way that data is leveraged in the system. One of the first casualties of the transition to an API-centric architecture is the practice of ETL (extract, transform, load).

In a pre-API architecture, many organizations find it necessary to have teams dedicated to extracting data by reverse engineering an app's internal structures and accessing databases directly, bypassing application logic. In an API-centric architecture, every app is responsible for exposing its data in a structured way via APIs. The extraction phase can be eliminated and transformation is greatly simplified as a result. Another common practice is to provide an analytics API to the application developer; they can push data to it in the appropriate places in their code. Often times, both techniques will be used, allowing analytics systems to easily harvest or pull data from various applications out of their APIs as well as enabling the individual applications to push data into the analytics system.

One aspect of the API-centric architecture that is particular relevant to discussions of data is related to the movement of interaction functionality into the client tier, as is the case with HTML5 and mobile. This clean separation of concerns means that most user interactions occur over the API channel rather than within web page presentation. As a consequence, the metadata about context, such as user identity and interaction intent, can be derived by observing and inspecting the API traffic.

Integration-centric architectures only cared about connecting systems together, and in most cases context was lost. However, the API-centric architecture allows context to be captured for business and operational purpose; a stream of contextual data can be sent to business intelligence systems and the signaling for operational monitoring systems helps ensure the overall health of the systems.

Taking this one step further, this contextual data can be used to drive context-aware applications, enabling the complete feedback loop. Personalization and recommendations are common examples of this, as are decision-support dashboards utilized by customer service personnel.

Context-Aware and Predictive Applications
An API-centric architecture leverages the API-based pull and push of application data to feed predictive intelligence engines, which communicate back to the application via APIs to drive actions.

To make this happen, the API-centric architecture leverages the pull and push of application data to feed predictive intelligence engines that then communicate back to the application via APIs. With an integration-based architecture, wiring up this kind of feedback loop would be cumbersome at best.

But with an API-centric architecture, applications can easily deliver the necessary data to the predictive analytics system and application developers can very easily leverage the resulting feedback to build individualized user experiences by constructing and presenting UX elements tailored to the specific user.

The API Architecture is the New Application Architecture

As business drives the demand for contextually-aware, highly personalized, predictive applications, delivered to new types of clients, and which are built in tighter and tighter timeframes and deployed at ever higher levels of scale, the application architecture has to move beyond the integration server/application-server pattern that has characterized much of the last decade of web application development.

API Architecture: The New Application Architecture

Build applications in an **agile** fashion

Deploy applications at **scale**

Future-proof for front-end technology

Connect to every other application inside and outside the **enterprise**

Personalize and recommend with context-aware and **predictive** applications

Applications must embrace the four-sided model of API architecture—app-to-client, app-to-backend, app-to-app, and the exploded app built from micro-service APIs.

Once this happens, then not only can the application be built in an agile fashion, deployed at scale, and support any form of future front-ends, it can also easily be connected to every other application inside and outside the enterprise. It can also easily share the relevant data with analytics systems, and in turn, deliver back data-driven, contextually-relevant actions based on real-time feedback loops driven from those same analytics systems.

Conclusion

As expectations of apps reach new heights and businesses are challenged with providing the right content and capabilities at just the right moment for the right person on any number of devices, application architecture has to move beyond the integration server/application-server pattern formed during the last decade of web application development.

Rather than using web frameworks that invoke services and produce web pages, today's applications are built by consuming and producing APIs. Applications have embraced APIs on the front end for connecting to rich clients; on the backend for integrating with internal systems; on the sides for enabling other applications to access their internal data and processes; and within as applications are composed of a set of component services, linked together with APIs.

An API-centric architecture has several important implications for today's enterprise:

- **Built for agility, scale and communication**
  An API- centric architecture enables applications to be built in an agile fashion, future-proofed for new front-end technology, deployed at scale, and easily connected to other applications and systems inside and outside the enterprise.

- **Demise of the centralized service governance model**
  The rise of virtualization, IaaS, and PaaS as well as a generation of internet developers with easy access to server resources, have all led to the demise of the centralized service governance model. SOA governance, which focused on centralized IT resources, has ceded ground to API governance, which focuses on supporting the application teams and agile and decentralized API-first architectures.

- **Integration models rooted in appliance-heritage products have no place in the automation-centric DevOps processes**
  New API use cases (especially those driven by mobile), as well as new API-centric and micro-service based development efforts, typically have little need for integration server technologies. As a result, heavyweight integration products have given way to a model where resources (including APIs and the systems they connect) are managed within the application tier by a set of DevOps automation tools designed for today's agile enterprise.

- **APIs make agile data possible and ETL obsolete**
  In an API-centric architecture, ETL (extract, transform, load) becomes obsolete. Instead, every app is responsible for exposing its data in a structured way via APIs. Furthermore, the API-based pull and push of application data enables a complete feedback loop, and feeds predictive intelligence engines, which can communicate back to the application via APIs to drive actions.

Enterprises can no longer afford to view APIs as simply an extension and evolution of integration-based architectures that have long been in use within enterprise IT. APIs and API-centric architecture have become the foundational technology necessary for the development and deployment of robust and scalable enterprise applications.

apigee

About Apigee

Apigee provides the leading technology platform for digital acceleration. Through APIs and big data, Apigee delivers the scale, insight, and agility any business needs to compete in today's digital world. Apigee customers include global enterprises such as Walgreens, eBay, Shell, Live Nation, Kaiser Permanente, and Sears. To learn more, go to apigee.com.

About the Author

Ed Anuff is a leader in product and technology strategy at Apigee with direct responsibility for mobile and developer products. A respected technologist, a proven innovator, and an experienced entrepreneur, Ed has designed and created innovative consumer and enterprise products, defined product strategy at early-stage and publicly traded companies, and founded and sold several technology companies.

Share this eBook